Advanced Real-Time Rendering in 3D Graphics and Games

SIGGRAPH 2007 Course 28

August 8, 2007

Course Organizer: Natalya Tatarchuk, AMD

Lecturers:

Johan Andersson, DICE
Shannon Drone, Microsoft
Nico Galoppo, UNC
Chris Green, Valve
Chris Oat, AMD
Jason L. Mitchell, Valve
Martin Mittring, Crytek GmbH
Natalya Tatarchuk, AMD

About This Course

Advances in real-time graphics research and the increasing power of mainstream GPUs has generated an explosion of innovative algorithms suitable for rendering complex virtual worlds at interactive rates. This course will focus on the interchange of ideas from game development and graphics research, demonstrating converging algorithms enabling unprecedented visual quality in real-time. This course will focus on recent innovations in real-time rendering algorithms used in shipping commercial games and high end graphics demos. Many of these techniques are derived from academic work which has been presented at SIGGRAPH in the past and we seek to give back to the SIGGRAPH community by sharing what we have learned while deploying advanced real-time rendering techniques into the mainstream marketplace.

This course was introduced to SIGGRAPH community last year and it was extremely well received. Our lecturers have presented a number of innovative rendering techniques – and you will be able to find many of those techniques shine in the upcoming state-of-the-art games shipping this year, and even see the previews of those games in this year's Electronic Theater.

This year we will bring an entirely new set of techniques to the table, and even more of them are coming directly from the game development community, along with industry and academia presenters. The second year version of this course will include state-of-the-art real-time rendering research as well as algorithms implemented in several award-winning games and will focus on general, optimized methods applicable in variety of applications including scientific visualization, offline and cinematic rendering, and game rendering. Some of the topics covered will include rendering face wrinkles in real-time; surface detail maps with soft self-shadowing and fast vector texture maps rendering in Valve's *Source*™ engine; interactive illustrative rendering in Valve's *Team Fortress 2*. This installation of the course will cover terrain rendering and shader network design in the latest *Frostbite* rendering engine from DICE, and the architectural design and framework for direct and indirect illumination from the upcoming *CryEngine 2.0* by Crytek. We will also introduce the idea of using GPU for direct computation of non-rigid body deformations at interactive rates, along as with advanced particle dynamics using DirectX10 API.

We will provide an updated version of these course notes with more materials about real-time tessellation and noise computation on GPU in real-time, downloadable from ACM Digital Library and from AMD ATI developer website prior to SIGGRAPH.

Prerequisites

This course is intended for graphics researchers, game developers and technical directors. Thorough knowledge of 3D image synthesis, computer graphics illumination models, the DirectX and OpenGL API Interface and high level shading languages and C/C++ programming are assumed.

Topics

Examples of practical real-time solutions to complex rendering problems:

- Terrain rendering with procedural texture splatting
- Real-time tessellation and noise generation on GPU
- Architectural design and illumination techniques from *CryEngine 2.0*
- Facial wrinkles rendering and animation
- Real-time particle systems on the GPU in dynamic environments
- GPU-accelerated simulation of deformable models in contact
- Efficient self-shadowed radiosity normal mapping
- Improved alpha-tested magnification for vector textures and special effects
- Illustrative rendering in *Team Fortress 2*

Suggested Reading

- Real-Time Rendering by Tomas Akenine-Möller, Eric Haines, A.K. Peters, Ltd.; 2nd edition, 2002
- Advanced Global Illumination by Philip Dutre, Phillip Bekaert, Kavita Bala, A.K. Peters, Ltd.; 1st edition, 2003
- Radiosity and Global Illumination by François X. Sillion, Claude Puech; Morgan Kaufmann, 1994.
- Physically Based Rendering : From Theory to Implementation by Matt Pharr, Greg Humphreys; Morgan Kaufmann; Book and CD-ROM edition (August 4, 2004)
- The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics, Steve Upstill, Addison Wesley, 1990.
- Advanced RenderMan: Creating CGI for Motion Pictures, Tony Apodaca & Larry Gritz, Morgan-Kaufman 1999.
- Texturing and Modeling, A Procedural Approach Second Edition, Ebert, Musgrave, Peachey, Perlin, Worley, Academic Press Professional, 1998.
- ShaderX5: Advanced Rendering Techniques, by Wolfgang Engel (Editor), Charles River Media, 1st edition (December 2006)
- ShaderX4: Advanced Rendering Techniques, by Wolfgang Engel (Editor), Charles River Media, 1st edition (November 2005)
- ShaderX3: Advanced Rendering with DirectX and OpenGL, by Wolfgang Engel (Editor), Charles River Media, 1st edition (November 2004)
- ShaderX2: Introductions and Tutorials with DirectX 9.0, by Wolfgang Engel (Editor), Wordware Publishing, Inc.; Book and CD-ROM edition (November 2003)
- ShaderX2 : Shader Programming Tips and Tricks with DirectX 9.0, by Wolfgang Engel (Editor), Wordware Publishing, Inc.; Book and CD-ROM edition (November 2003)

Lecturers

Natalya Tatarchuk is a staff research engineer leading the research team in AMD's 3D Application Research Group, where pushes the GPU boundaries investigating innovative graphics techniques and creating striking interactive renderings leading the research team. In the past she led the creation of the state-of-the-art realistic rendering of city environments in ATI demo “ToyShop” and has been the lead for the tools group at ATI Research. Natalya has been in the graphics industry for years, having previously worked on haptic 3D modeling software, scientific visualization libraries, among others. She has published multiple papers in various computer graphics conference and articles in technical book series such as ShaderX and Game Programming Gems, and has presented talks at Siggraph and at Game Developers Conferences worldwide, amongst others. Natalya holds BAs in Computers Science and Mathematics from Boston University.

Chris Oat is a staff engineer in AMD's 3D Application Research Group where he is the technical lead for the group's demo team. In this role, he focuses on the development of cutting-edge rendering techniques for leading edge graphics platforms. Christopher has published several articles in the ShaderX and Game Programming Gems series and has presented his work at graphics and game developer conferences around the world.

Jason L. Mitchell is a software developer at Valve, where he works on real-time graphics techniques for all of Valve's projects. Prior to joining Valve in 2005, Jason worked at ATI for 8 years, where he led the 3D Application Research Group. He received a BS in Computer Engineering from Case Western Reserve University and an MS in Electrical Engineering from the University of Cincinnati.

Chris Green is a software engineer at Valve, and has working on the Half-Life 2 series and Day of Defeat. Prior to joining Valve, Chris Green worked on such projects as Flight Simulator II, Ultima Underworld, the Amiga OS, and Magic:The Gathering Online. He ran his own development studio, Leaping Lizard Software, for 9 years.

Johan Andersson is a self-taught senior software engineer/architect in the central technology group at DICE. For the past 7 years he has been working on the rendering and core engine systems for games in the RalliSport and Battlefield series. He now drives the rendering side of the new Frostbite engine for the pilot game Battlefield: Bad Company (Xbox 360, PS3). Recent contributions include a talk at GDC about graph-based procedural shading.

Martin Mittring is a software engineer and member of the R&D staff at Crytek. Martin started his first experiments early with text-based computers, which led to a passion for computer and graphics in particular. He studied computer science and worked in one other German games company before he joined Crytek. During the development of Far Cry he was working on improving the Polybump™ tools and was became lead network programmer for that game. His passion for graphics brought him back to former path and so he became lead graphics programmer in R&D. Currently he is busy working on the next iteration of the engine to keep pushing future PC and next-gen console technology.

Nico Galoppo is currently a PhD. student in the GAMMA research group at the UNC Computer Science Department, where his research is mainly related to physically based animation and simulation of rigid, quasi-rigid and deformable objects, adaptive dynamics of articulated bodies, hair rendering, and many other computer graphics related topics. He also has experience with accelerated numerical algorithms on graphics processors, such as matrix decomposition. His advisor is Prof. Ming C. Lin and he is also in close collaboration with Dr. Miguel A. Otaduy

(ETHZ). Nico has published several peer-reviewed papers in various ACM conference proceedings had he has presented his work at the SIGGRAPH and ACM Symposium of Computer Animation conferences. Nico grew up in Belgium and holds an MSc in Electrical Engineering from the Katholieke Universiteit Leuven.

Shanon Drone is a software developer at Microsoft. Shanon joined Microsoft in 2001 and has recently been working on Direct3D 10 samples and applications. He spends a great deal of his time researching and implementing new and novel graphics techniques.

Contents

Preface

Welcome to the Advanced Real-Time Rendering in 3D Graphics and Games course at SIGGRAPH 2007. We've included both 3D Graphics and Games in our course title in order to emphasize the incredible relationship that is quickly growing between the graphics research and the game development communities. Although in the past interactive rendering was synonymous with gross approximations and assumptions, often resulting in simplistic visual rendering, with the amazing evolution of the processing power of consumer-grade GPUs, the gap between offline and real-time rendering is rapidly shrinking. Real-time domain is now at the forefront of state-of-the-art graphics research – and who wouldn't want the pleasure of instant visual feedback?

As researchers, we focus on pushing the boundaries with innovative computer graphics theories and algorithms. As game developers, we bend the existing software APIs such as DirectX and OpenGL and the available hardware to perform our whims at highly interactive rates. And as graphics enthusiasts we all strive to produce stunning images which can change in a blink of an eye and let us interact with them. It is this synergy between researchers and game developers that is driving the frontiers of interactive rendering to create truly rich, immersive environments. There is no greater satisfaction for developers than to share the lessons learned and to see our technologies used in ways never imagined.

This is the second time this course is presented at SIGGRAPH and we hope that you enjoy the new material presented this year and come away with a new understanding of what is possible without sacrificing interactivity! We hope that we will inspire you to drive the real-time rendering research and games!

Natalya Tatarchuk, AMD
April, 2007

Chapter 1

Efficient Self-Shadowed Radiosity Normal Mapping

Chris Green[1]

VALVE

Normal Mapped

Normal mapped with ambient occlusion

Self-shadowed

Figure 1. *Comparison of surface illumination techniques.*

1.1 Abstract

In Valve's *Source* graphics engine, bump mapping is combined with precomputed radiosity lighting to provide realistic surface illumination. When bump map data is derived from geometric descriptions of surface detail (such as height maps), only the lighting effects caused by the surface orientation are preserved. The significant lighting cues due to lighting occlusion by surface details are lost. While it is common to use another texture channel to hold an "ambient occlusion" field, this only provides a darkening effect which is independent of the direction from which the surface is being lit and requires an auxiliary channel of data.

In this chapter, we present a modification to the *Radiosity Normal Mapping* system that we have described in this course in the past. This modification provides a directional occlusion function to the bump maps, which requires no additional texture memory and is faster than our previous non-shadowing solution.

[1] email: cgreen@valvesoftware.com

1.2 Introduction

In order to increase surface detail and perceived realism, bump mapping is used heavily in modern real-time 3D games [Blinn78] [PAC97]. Bump maps are often used as an approximation of higher detailed geometry which would either be to slow to render in real time or too memory intensive to store reasonably. However, one weakness of bump maps is that they only modify the surface normal which is used for lighting computations. While this provides a realistic directional lighting cue, effects such as self-shadowing of surface details and ambient occlusion are not rendered.

Traditional bump mapping also cannot be combined with conventional precomputed light maps, a technique in which a global illumination solution is generated as a precomputation and then stored in a low resolution texture which is used to modulate the brightness of the surface texture [ID97].

With *Radiosity Normal Mapping*, the precomputed light map information was extended to encompass lighting from multiple directions and allowed arbitrary bump mapped data to be combined with precomputed lighting textures [McTaggart04] [MMG06]. Using Radiosity Normal Mapping, the distribution of incoming distributed lighting can be stored in many possible bases [Sloan06], with the tangent-space surface normal evaluated per pixel and then convolved with the incoming light distribution for each output pixel.

In this presentation, we extend Radiosity Normal Mapping by modifying the bump map representation to be pre-convolved against the tangent-space lighting basis in order to gain efficiency and reduce rendering artifacts. We then extend this pre-convolution to take into account directional self-occlusion information, allowing for both occlusion of isotropic ambient lighting and dynamic directional lighting.

1.3 Related Work

Many different techniques have been used to add shadowing information to bump mapped surfaces for real-time rendering. Horizon mapping augments bump map data by precomputing and storing the angle to the “horizon” for a small set of fixed tangent-space directions and uses this representation to produce hard shadows [Max98] [SC00]. In [KHD00], an oriented ellipse is fitted to the distribution of non-shadowed lights over a bump map texel. In [OS07], a spherical cap is used to model the visible light directions over a bump map texel, and this data is used to render hard and soft shadows from point and area lights in real time.

Recently, techniques have been developed for direct rendering of height fields in real time using graphics hardware [POC05] [MM05] [Tatarchuk06]. Since these techniques are able to compute visibility of height field texels from any viewpoint, they are also able to implement shadowing of height fields by computing visibility to light sources.

1.4 Representation and Generation

We wished to implement self-shadowing of bump maps that would

- Mesh well with our existing radiosity-lit bump map approach
- Work on older generations of graphics hardware as well as current systems.
- Run as fast as, or faster than our current non-shadowed solution.
- Improve bump map anti-aliasing
- Work with dynamics lights as well as our pre-calculated radiosity lighting
- Provide soft shadows and ambient occlusion
- Allow shadowing information to be generated either from height data or from arbitrary geometry
- Not use any increased texture storage space compared to ordinary bump mapping. In particular, we wanted to be able to preserve existing uses of the alpha channel of bump maps as a mask for various effects.

We successfully implemented a method to generate diffuse soft shadows from static and dynamic light sources, with no increase in bump map storage space, while providing an actual performance increase over our existing non-shadowed radiosity bump map implementation. The source engine calculates lighting at each surface pixel using the following operations: `normal = 2.0 * normalTexel - 1.0`.

$$\vec{B}_{0..2} = \left\{ -\frac{1}{\sqrt{6}}, \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{3}} \right\}, \left\{ -\frac{1}{\sqrt{6}}, \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{3}} \right\}, \left\{ -\frac{\sqrt{2}}{\sqrt{3}}, 0, \frac{1}{\sqrt{3}} \right\} \quad (1)$$

$$\vec{N} = 2 \times \vec{T} - 1 \quad (2)$$

$$\vec{D}_i = \frac{saturate(\vec{N} \cdot \vec{B}_i)^2}{\sum_{i-0}^{2} \vec{D}_i} \quad (3)$$

$$pixelcolor = albedo \times \sum_{i=0}^{2} \vec{L}_i \vec{D}_i \quad (4)$$

where $\vec{T}$ is the tangent-space bump map texel (which must be scaled and biased before use, because texels are unsigned), $\vec{B}$ is the set of tangent-space directions for which incoming light has been precomputed, $\vec{L}$ are the 3 precomputed lighting values, and $\vec{albedo}$ is the color of the surface texel being lit. `Saturate(x)` is the HLSL function which clamps its input to be in the range between 0 and 1.

```
float3 normal = 2.0 * normalTexel - 1.0;
float3 dp;
dp.x = saturate( dot( normal, bumpBasis[0] ) );
dp.y = saturate( dot( normal, bumpBasis[1] ) );
dp.z = saturate( dot( normal, bumpBasis[2] ) );
dp *= dp;

float sum = dot( dp, float3( 1.0f, 1.0f, 1.0f ) );
float3 diffuseLighting = dp.x * lightmapColor1 +
                         dp.y * lightmapColor2 +
                         dp.z * lightmapColor3;
diffuseLighting /= sum;
```

Listing 1. *HLSL source code for original* Source *radiosity bumpmapping.*

We observed that some execution time could be saved if, instead of storing the surface normal, $\vec{N}$ in our texture maps, we instead stored the value of $\vec{D}$ in the 3 color components of the bump map. This is a trivial modification to any bump map generation program. This reduces the lighting equation to equation (4) above.

```
float3 diffuseLighting = normalTexel.x * lightmapColor1 +
                         normalTexel.y * lightmapColor2 +
                         normalTexel.z * lightmapColor3;
```

Listing 2. *HLSL source code for new directional-ambient-occlusion bump mapping*

Just doing this saves a substantial number of pixel shader instructions. However, we no longer have a tangent-space surface normal for use to calculate the lighting from dynamic lights and reflection vectors. Nonetheless, when needed, we can use our original basis directions $\vec{B}$ to reconstruct a suitable tangent-space normal for reflections. For dynamic lights, we can project the lighting direction onto our basis directions and use that directly in the lighting equation, which gives us a form of shadowing from dynamic light sources.

Once bump maps are stored in this format, some advantages are seen besides the increased pixel shader performance:

- Because the bump maps now just represent positive light map texture blending weights, no special processing is required when filtering or mip-mapping them.
- Numeric precision is increased for the radiosity bump mapped case, since we are now storing the bump maps in the exact numeric representation that their data is needed in, and since we do not have to represent negative numbers.
- Surface textures stored in this form can be processed by existing art tools with no special interpretation. For instance, filters such as sharpen and blur can be used in Photoshop™.
- Texture blending for operations such as detail texturing, texture cross fading, etc. are much more straightforward.
- Fewer aliasing artifacts will be seen when textures are minimized.

- Textures in this format can be directly generated from geometry in 3D rendering packages, by placing red/green/blue unshadowed point lights in the scene, offset along the 3 predefined basis vectors.

Ordinary bump maps can only change the apparent lighting orientation of the surface. However, when rendering with this representation, if we uniformly scale the RGB values of the normal map texels, we can provide a darkening effect which is independent of the lighting direction. This allows us to have normal maps also act to modulate surface albedo, without having to store a separate brightness channel, or change the RGB values of the base albedo texture. Since it is often possible to produce good imagery by combining a fairly low frequency albedo texture with a high frequency bump map texture, this can save us texture memory.

A common bump map production method involves taking elevation maps which are either painted, created in a modeling package, or acquired from real-word sources, and then using these elevation maps to extract a surface normal. Then, the same elevation data is used to calculate an ambient occlusion channel, which is typically generated by firing from each texel, a large set of rays. The results of these ray intersections are used to determine the cosine-weighted proportion of the hemisphere above the surface which can be seen without the surface obscuring it. The result of this ambient occlusion calculation is then either stored in its own texture channel, or multiplied into the base texture. We can do the exact same thing in our representation, except that *we can encode this channel directly into the 3 channel normal map*.

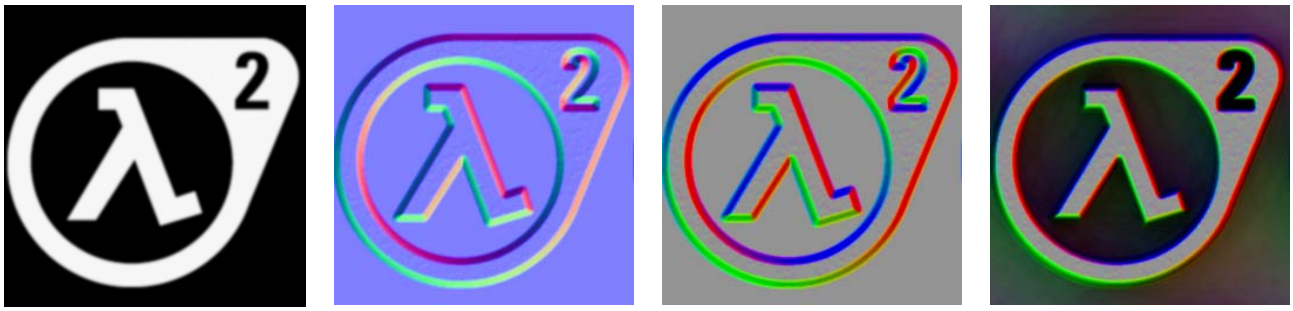


Figure 2. *(a) Height map (b) Bump map (c) Bump map stored in our basis (d) With directional ambient occlusion*

Moreover, since each of the channels of our modified bump map store the amount of light coming from 3 fixed tangent-space directions, we are able to do better than just encoding ambient occlusion. If we weight the contribution of each ray intersection test based upon the dot product between it and the fixed tangent-space direction associated with each channel, we can calculate a separate occlusion value for each channel, with that occlusion value representing a smaller angular distribution centered upon the corresponding basis vector. This gives us a “directional ambient occlusion” effect which causes ambient and direct lighting arriving from different directions to be darkened when that light would have been blocked by the self-shadowing effects of the surface. These 3 ambient occlusion directions are simply multiplied into the 3 non-shadowed bump maps we are already using in our representation. This gives us form of diffuse self-shadowing

essentially “for free”, providing self-shadowing of direct and indirect light. When these directional ambient occlusion textures are converted back into normal vectors for reflection calculations, something akin to the use of “bent normals” is achieved.

We implemented an efficient multi-threaded SIMD ray tracing system [WBW+01] in order to perform the hundreds of ray intersection tests per texel necessary to generate accurate direction ambient occlusion. Our off line generation utility takes as input an elevation map and an elevation scale factor. A user-configurable bilateral filter [TM98] is applied to the input image to reduce stair-stepping, and then 300 rays per output texel are traced in order to generate bump maps with directional ambient occlusion in our new format.

It is also possible to generate textures in this format through standard 3D rendering packages, by the careful placement of area lights in the scenes. Analogously, such maps could be captured from real world materials via photography with appropriately placed lights and reflectors.

The standard techniques for generating bump maps for a coarsely tessellated model by tracing rays against a more finely tessellated one can be easily extended to support this bump map representation, which will allow for animated articulated models with self-shadowing surface detail.

We can easily extend this technique to support more channels in order to produce more accurate lighting and shadows, at the expense of higher texture storage. Differing combinations of sample directions and lighting precomputation directions can be used, for instance to provide more accurate shadows from dynamic lights without increasing the storage needed for light maps.

Figure 3. *Cave walls exhibiting self-shadowing from the flashlight in Half-Life®2: Episode 2.*

1.5 Conclusion

A form of self-shadowing can be easily added to radiosity bump mapping by “baking” the light sampling basis into the actual bump map data, with no decrease in performance or increase in texture memory cost. We are able to make heavy use of this technique in the games Half-Life® 2: Episode 2 and Team Fortress 2.

1.6 References

[BLINN78] BLINN, J. F. 1978. Simulation of wrinkled surfaces. In SIGGRAPH ’78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques, ACM Press, New York, NY, USA, 286–292.

[ID97] ID SOFTWARE, 1997. Quake 2.

[KHD00] KAUTZ, J., HEIDRICH, W., AND DAUBERT, K. 2000. Bump map shadows for OpenGL rendering. Tech. Rep. MPI-I-2000-4-001, Max-Planck-Institut für Informatik, Saarbrücken, Germany.

[MAX98] MAX, N. L. 1998. Horizon mapping: shadows for bump-mapped surfaces. In The Visual Computer, 109–117.

[MM05] MCGUIRE, M., AND MCGUIRE, M. 2005. Steep parallax mapping. I3D 2005 Poster.

[MCTAGGART04] MCTAGGART, G., 2004. Half-life 2 shading. GDC Direct3D Tutorial.

[MMG06] MITCHELL, J., MCTAGGART, G., AND GREEN, C. 2006. Shading in Valve’s source engine. In SIGGRAPH ’06: ACM SIGGRAPH 2006 Courses, ACM Press, New York, NY, USA, 129–142.

[OS07] OAT, C., AND SANDER, P. V. 2007. Ambient aperture lighting. In I3D ’07: Proceedings of the 2007 symposium on Interactive 3D graphics and games, ACM Press, New York, NY, USA, 61–64.

[PAC97] PEERCY, M., AIREY, J., AND CABRAL, B. 1997. Efficient bump mapping hardware. Computer Graphics 31, Annual Conference Series, 303–306.

[POC05] POLICARPO, F., OLIVEIRA, M. M., AND COMBA, J. L. D. 2005. Real-time relief mapping on arbitrary polygonal surfaces. In I3D ’05: Proceedings of the 2005 symposium on Interactive 3D graphics and games, ACM Press, New York, NY, USA, 155–162.

[SC00] SLOAN, P.-P. J., AND COHEN, M. F. 2000. Interactive horizon mapping. In Proceedings of the Eurographics Workshop on Rendering Techniques 2000, Springer-Verlag, London, UK, pp. 281–286.

[SLOAN00] SLOAN, P.-P. 2006. Normal mapping for precomputed radiance transfer. In I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games, ACM Press, New York, NY, USA, pp. 23–26.

[TATARCHUK06] TATARCHUK, N. 2006. Dynamic parallax occlusion mapping with approximate soft shadows. In proceedings of AMD SIGGRAPH Symposium on Interactive 3D Graphics and Games, pp. 63-69, Redwood City, CA.

[TM98] TOMASI, C., AND MANDUCHI, R. 1998. Bilateral filtering for gray and color images. In ICCV, 839–846.

[WBW+01] WALD, I., BENTHIN, C., WAGNER, M., AND SLUSALLEK, P. 2001. Interactive rendering with coherent ray tracing. Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001), v. 20, i. 3, pp. 153-164.

Chapter 2

Improved Alpha-Tested Magnification for Vector Textures and Special Effects

Chris Green[2]

VALVE

(a) 64x64 texture, alpha-blended *(b) 64x64 texture, alpha tested* *(c) 64x64 texture using our technique*

Figure 1. *Vector art encoded in a 64x64 texture using (a) simple bilinear filtering (b) alpha testing and (c) our distance field technique*

2.1 Abstract

A simple and efficient method is presented which allows improved rendering of glyphs composed of curved and linear elements. A distance field is generated from a high resolution image, and then stored into a channel of a lower-resolution texture. In the simplest case, this texture can then be rendered simply by using the alpha-testing and alpha-thresholding feature of modern GPUs, without a custom shader. This allows the technique to be used on even the lowest-end 3D graphics hardware.

[2] email: cgreen@valvesoftware.com

With the use of programmable shading, the technique is extended to perform various special effect renderings, including soft edges, outlining, drop shadows, multi-colored images, and sharp corners.

2.2 Introduction

For high quality real-time 3D rendering, it is critical that the limited amount of memory available for the storage of texture maps be used efficiently. In interactive applications such as computer games, the user is often able to view texture mapped objects at a high level of magnification, requiring that texture maps be stored at a high resolution so as to not become unpleasantly blurry, as shown in Figure 1a, when viewed from such perspectives.

When the texture maps are used to represent “line-art” images, such as text, signs and UI elements, this can require the use of very high resolution texture maps in order to look acceptable, particularly at high resolutions.

In addition to text and UI elements, this problem is also common in alpha-tested image-based impostors for complicated objects such as foliage. When textures with alpha channels derived from coverage are magnified with hardware bilinear filtering, unpleasant “wiggles” as seen in Figure 1b appear because the coverage function is not linear.

In this chapter, we present a simple method to generate and render alpha-tested texture maps in order to minimize the artifacts that arise when such textures are heavily magnified. We will demonstrate several usage scenarios in a computer game context, both for 3D renderings and also user-interface elements. Our technique is capable of generating high quality vector art renderings as shown in Figure 1c.

2.3 Related Work

Many techniques have been developed to accurately render vector graphics using texture-mapping graphics hardware. In [FPR+00], distance fields were used to represent both 2-dimensional glyphs and 3-dimensional solid geometry. Quadtrees and octrees were used to adaptively control the resolution of the distance field based upon local variations. While GPU rendering of such objects was not discussed, recent advances in the generality of GPU programming models would allow this method to be implemented using DirectX10 [Blythe06].

In [Sen04] and [TC04], texture maps are augmented with additional data to control interpolation between texel samples so as to add sharp edges in a controllable fashion. Both line-art images and photographic textures containing hard edges were rendered directly on the GPU using their representation. In [LB05], implicit cubic curves were used to model the boundaries of glyphs, with the GPU used to render vector textures with smooth resolution-independent curves. In [QMK06], a distance based representation is

used, with a precomputed set of “features” influencing each Voronoi region. Given these features, a pixel shader is used to analytically compute exact distance values.

Alpha-testing, in which the alpha value output from the pixel shader is thresholded so as to yield a binary on/off result, is widely used in games to provide sharp edges in reconstructed textures. Unfortunately, because the images that are generally used as sources for this contain “coverage” information which is not properly reconstructed at the sub-texel level by bilinear interpolation, unpleasant artifacts

2.4 Representation and Generation

In order to overcome the artifacts of simple alpha testing while keeping storage increase to a minimum, we sought a method for displaying vector textures that could

- Work on all levels of graphics hardware, including systems lacking programmable shading
- Run as fast as, or nearly as fast as, standard texture mapping
- Take advantage of the bilinear interpolation present in all modern GPUs
- Function inside of a pre-existing complex shader system [MMG06] with few changes
- Add at most a few instructions to the pixel shader so that vector textures can be used in existing shaders without overflowing instruction limits
- Not require that input images be provided in a vector form
- Use existing low-precision 8-bit texture formats
- Be used as a direct replacement for alpha-tested impostor images

We chose to implement a simple uniformly-sampled signed distance field representation, with the distance function stored in an 8-bit channel. By doing so, we are able to take advantage of the native bilinear texture interpolation which is present in all modern GPUs in order to accurately reconstruct the distance between a sub-texel and a piecewise-linear approximation of the true high resolution image. While this representation is limited in terms of the topology of features which can be represented compared to other approaches, we felt that its high performance, simplicity, and ease of integration with our existing rendering system made it the right choice for Valve’s *Source* engine.

While it is possible to generate the appropriate distance data from vector-based representations of the underlying art, we choose instead to generate the low-resolution distance fields from high resolution source images. In a typical case, a 4096×4096 image will be used to generate a distance field texture with a resolution as low as 64×64, as shown in Figure 2.

(a) High resolution input

(b) 64×64 *Distance field*

Figure 2*. (a) A high resolution (*4096×4096*) binary input is used to compute (b) a low resolution (*64×64*) distance field*

At texture-generation time, the generator takes as its input a high resolution binary texture where each texel is classified as either “in” or “out.” The user specifies a target resolution, and also a “spread factor,” which controls the range which is used to map the signed distance into the range of 0 to 1 for storage in an 8-bit texture channel. The spread factor also controls the domain of effect for such special rendering attributes as drop-shadows and outlines, which will be discussed in Section 2.5.2.

For each output texel, the distance field generator determines whether the corresponding pixel in the high resolution image is “in” or “out.” Additionally, the generator computes 2D distance (in texels) to the nearest texel of the opposite state. This is done by examining the local neighborhood around a given texel. While there are more efficient and complex algorithms to compute the signed distance field than our simple “brute-force” search, because of the limited distance range which may be stored in an 8-bit alpha channel, only a small neighborhood must be searched. The execution time for this simple brute-force method is negligible.

Once this signed distance has been calculated, we map it into the range 0..1, with 0 representing the maximum possible negative distance and 1.0 representing the maximum possible positive distance. A texel value of 0.5 represents the exact position of the edge and, hence, 0.5 is generally used for the alpha threshold value.

2.5 Rendering

In the simplest case, the resulting distance field textures can be used as-is in any context where geometry is being rendered with alpha-testing. Under magnification, this will produce an image with high-resolution (albeit, aliased) linear edges, free of the false curved contours (see Figure 1b) common with alpha-tested textures generated by storing and filtering coverage rather than a distance field. With a distance field representation, we merely have to set the alpha test threshold to 0.5. Since it is fairly common to use alpha testing rather than alpha blending for certain classes of primitives

Figure 3. 128×128 *"No trespassing" distance image applied to a surface in Team Fortress 2*

in order to avoid the costly sorting step, this technique can provide an immediate visual improvement with no performance penalty.

In Figure 3, we demonstrate a 128×128 distance field representation of a "No Trespassing" sign rendered as a decal over the surface of a wall in the game *Team Fortress 2*. The apparent resolution of this decal is incredibly high in world space and holds up well under any level of magnification that it will ever undergo in the game. We will refer to this particular decal example in the next section as we discuss other enhancements available to us when representing our vector art in this manner.

2.5.1 Antialiasing

If alpha-blending is practical for a given application, the same distance field representation can be used to generate higher quality renderings than mere alpha testing, at the expense of requiring custom fragment shaders.

Figure 4 demonstrates a simple way to soften up the harsh aliased pixel edges. Two distance thresholds, $Dist_{min}$ and $Dist_{max}$, are defined and the shader maps the distance field value between these two values using the `smoothstep()` function. On graphics hardware which supports per-pixel screen-space derivatives, the derivatives of the distance field's texture coordinates can be used to vary the width of the soft region in order to properly anti-alias the edges of the vector art [QMK06]. When the texture is minified, widening of the soft region can be used to reduce aliasing artifacts. Additionally, when rendering alpha-tested foliage, the alpha threshold can be increased with distance, so that the foliage gradually disappears as it becomes farther away to avoid LOD popping.

Figure 4. *Zoom of* 256×256 ``*No Trespassing'' sign with hard (left) and softened edges (right)*

2.5.2 Enhanced Rendering

In addition to providing crisp high resolution anti-aliased vector art using raster hardware, we can apply additional manipulations using the distance field to achieve other effects such as outlining, glows and drop shadows. Of course, since all of these operations are functions of the distance field, they can be dynamically controlled with shader parameters.

2.5.2.1 Outlining

By changing the color of all texels which are between two user-specified distance values, a simple texture-space outlining can be applied by the pixel shader as shown in our decal example in Figure 5. The outline produced will have crisp high quality edges when magnified and, of course, the color and width of the outline can be varied dynamically merely by changing pixel shader constants.

2.5.2.2 Glows

When the alpha value is between the threshold value of 0.5 and 0, the smoothstep function can be used to substitute a “halo” whose color value comes from a pixel shader constant as shown in Figure 6. The dynamic nature of this effect is particularly powerful in a game, as designers may want to draw attention to a particular piece of vector art in the game world based on some game state by animating the glow parameters (blinking a health meter, emphasizing an exit sign etc).

Figure 5*. Outline added by pixel shader*

Figure 6*. Scary flashing "Outer glow" added by pixel shader*

Figure 7*. Soft drop-shadow added by the pixel shader. The direction, size, opacity, and color of the shadow are dynamically controllable.*

2.5.2.3 Drop Shadows

In addition to effects which are simple functions of a single distance, we can use a second lookup into the distance field with a texture coordinate offset to produce drop shadows or other similar effects as shown in Figure 6. In addition to these simple 2D effects, there are surely other ways to reinterpret the distance field to give designers even more options.

2.5.3 Sharp Corners

As in all of the preceding examples, encoding edges using a single signed distance "rounds off" corners as the resolution of the distance field decreases [QMK06]. For example, the hard corners of the letter G in Figure 2a become more rounded off as illustrated in Figures 5, 6 and 7.

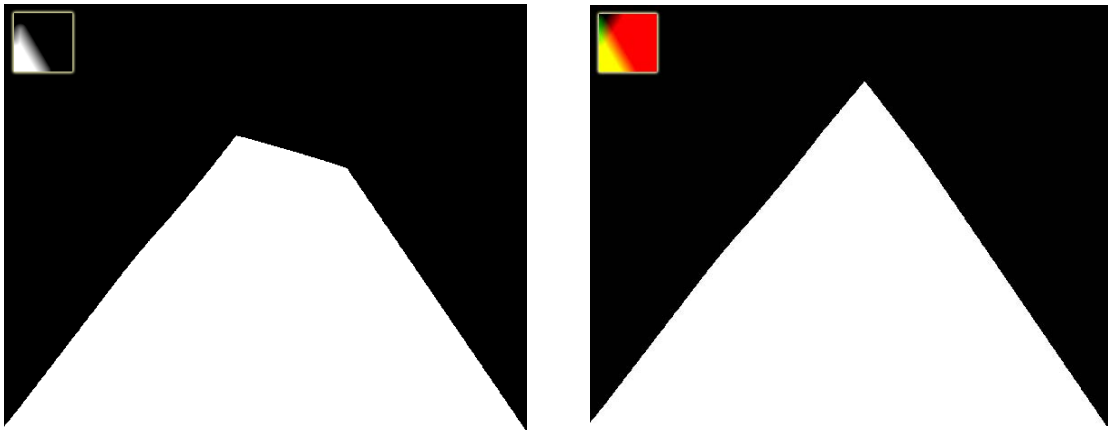


Figure 8*. Corner encoded at 64x64 using one distance field (left) and the AND of two distance fields (right)*

Sharp corners *can* be preserved, however, by using more than one channel of the texture to represent different edges intersecting within a texel. For instance, with two channels, the intersection of two edges can be accurately represented by performing a logical AND in the pixel shader. In Figure 8, we have stored these two edge distances in the red and green channels of a single texture, resulting in a well-preserved pointy corner. This same technique could also be performed on the “No Trespassing” sign if we wished to represent sharper corners on our text. As it stands, we like the rounded style of this text and have used a single distance field for this and other decals in *Team Fortress 2*.

2.7 Conclusion

In this chapter, we have demonstrated an efficient vector texture system which has been integrated into the *Source* game engine which has been previously used to develop games such as the *Half-Life® 2* series, *Counter-Strike: Source* and *Day of Defeat: Source*. This vector texture technology is used in the upcoming game *Team Fortress 2* with no significant performance degradation relative to conventional texture mapping. We were able to effectively use vector-encoded images both for textures mapped onto 3D geometry in our first person 3D view and also for 2D screen overlays. This capability has provided significant visual improvements and savings of texture memory.

2.8 References

[BLYTHE06] BLYTHE, D. 2006. The direct3d 10 system. In SIGGRAPH '06: ACM SIGGRAPH 2006 Papers, ACM Press, New York, NY, USA, pp. 724–734.

[FPR+00] FRISKEN, S. F., PERRY, R. N., ROCKWOOD, A. P., AND JONES, T. R. 2000. Adaptively sampled distance fields: a general representation of shape for computer graphics. In SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, pp. 249–254.

[LB05] LOOP, C., AND BLINN, J. 2005. Resolution independent curve rendering using programmable graphics hardware. In SIGGRAPH '05: ACM SIGGRAPH 2005 Papers, ACM Press, New York, NY, USA, pp. 1000–1009.

[MMG06] MITCHELL, J., MCTAGGART, G., AND GREEN, C. 2006. Shading in Valve's source engine. In SIGGRAPH '06: ACM SIGGRAPH 2006 Courses, ACM Press, New York, NY, USA, pp. 129–142.

[QMK06] QIN, Z., MCCOOL, M. D., AND KAPLAN, C. S. 2006. Real-time texture-mapped vector glyphs. In I3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games, ACM Press, New York, NY, USA, pp. 125–132.

[RNC+05] RAY, N., NEIGER, T., CAVIN, X., AND LEVY, B. 2005. Vector texture maps. In Tech Report.

[SEN04] SEN, P. 2004. Silhouette maps for improved texture magnification. In HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, ACM Press, New York, NY, USA, pp. 65–73.

[TC04] TUMBLIN, J., AND CHOUDHURY, P. 2004. Bixels: Picture samples with sharp embedded boundaries. In Rendering Techniques, pp. 255–264.

```
float distAlphaMask = baseColor.a;

if ( OUTLINE &&
    ( distAlphaMask >= OUTLINE_MIN_VALUE0 ) &&
    ( distAlphaMask <= OUTLINE_MAX_VALUE1 ) )
{
  float oFactor=1.0;
  if ( distAlphaMask <= OUTLINE_MIN_VALUE1 )
  {
    oFactor=smoothstep( OUTLINE_MIN_VALUE0,
                        OUTLINE_MIN_VALUE1,
                        distAlphaMask );
  }
  else
  {
    oFactor=smoothstep( OUTLINE_MAX_VALUE1,
                        OUTLINE_MAX_VALUE0,
                        distAlphaMask );
  }
  baseColor = lerp( baseColor, OUTLINE_COLOR, oFactor );
}

if ( SOFT_EDGES )
{
  baseColor.a *= smoothstep( SOFT_EDGE_MIN,
                             SOFT_EDGE_MAX,
                             distAlphaMask );
}
else
{
  baseColor.a = distAlphaMask >= 0.5;
}

if ( OUTER_GLOW )
{
  float4 glowTexel =
        tex2D( BaseTextureSampler,
               i.baseTexCoord.xy+GLOW_UV_OFFSET );

  float4 glowc = OUTER_GLOW_COLOR * smoothstep(
                   OUTER_GLOW_MIN_DVALUE,
                   OUTER_GLOW_MAX_DVALUE,
                   glowTexel.a );
  baseColor = lerp( glowc, baseColor, mskUsed );
}
```

Listing 1. *HLSL source code for outline, glow/drop shadow, and edge softness.*

Chapter 3

Illustrative Rendering in Team Fortress 2

Jason L. Mitchell[3] Moby Francke [4] Dhabih Eng [5]

VALVE

(a) Concept art

(b) Character in the game

Figure 1. *(a) Concept Art (b) Character as seen by players during gameplay*

The content of this chapter also appears on Symposium on Non-Photorealistic Animation and Rendering 2007.

[3] e-mail: jasonm@valvesoftware.com
[4] e-mail: moby@valvesoftware.com
[5] e-mail: dhabih@valvesoftware.com

3.1 Abstract

We present a set of artistic choices and novel real-time shading techniques which support each other to enable the unique rendering style of the game *Team Fortress 2*. Grounded in the conventions of early 20th century commercial illustration, the look of *Team Fortress 2* is the result of tight collaboration between artists and engineers. In this paper, we will discuss the way in which the art direction and technology choices combine to support artistic goals and gameplay constraints. In addition to achieving a compelling style, the shading techniques are designed to quickly convey geometric information using rim highlights as well as variation in luminance and hue, so that game players are consistently able to visually "read" the scene and identify other players in a variety of lighting conditions.

3.2 Introduction

We present a set of artistic choices and real-time shading techniques which support each other to enable the unique Non-Photorealistic Rendering (NPR) style of *Team Fortress 2*. Grounded in the conventions of early 20th century commercial illustration with 1960s industrial design elements, the look of *Team Fortress 2* is the result of close collaboration between artists and engineers. At Valve, we believe in having the two disciplines heavily influence each other and, in this paper, we will discuss the ways in which art and technology choices combine to support stylistic goals and gameplay constraints. While most non-photorealistic rendering techniques are presented on a single rigid model in no particular lighting environment, we will demonstrate a series of interactive rendering techniques which result in a cohesive and dynamic environment. Rather than merely achieving a stylized look, the shading techniques are designed to quickly convey geometric information in our desired illustrative style using variation in luminance and hue, so that game players are consistently able to visually "read" the scene and identify other players in a variety of lighting conditions.

For *Team Fortress 2*, the 2007 sequel to the popular *Half-Life* mod *Team Fortress Classic*, we sought to explicitly differentiate ourselves from other multiplayer deathmatch games which typically embrace a modern photorealistic look. In *Team Fortress 2*, we chose to employ an art style inspired by the early to mid 20th century commercial illustrators J. C. Leyendecker, Dean Cornwell and Norman Rockwell [Schau74]. These artists were known for illustrating characters using strong, distinctive silhouettes with emphasis on clothing folds and they tended to use shading techniques which accentuated the internal shape of objects and characters with patterns of value while emphasizing silhouettes with rim highlights rather than dark outlines, as shown in the concept art in Figure 1a.

Contributions of this paper include the codification of key conventions of the commercial illustrations of Leyendecker, Cornwell and Rockwell as well as methods for generating such renderings in a real-time game. Specific technical contributions include the implementation of a diffuse light warping function appropriate for illustrative rendering, a novel formulation of rim lighting and an overall balance of photorealistic and non-

photorealistic shading techniques to achieve the desired look while serving gameplay goals.

In the next section, we will discuss previous work which relates to ours. In Section 3.4, we will enumerate the specific properties common to commercial illustration which define our style. In Section 3.5, we will briefly discuss the creation of art for *Team Fortress 2*. In Section 3.6, we will discuss our shading algorithms in detail, before concluding with the topics of abstraction and future work.

3.3 Related Work

Non-photorealistic rendering styles can vary greatly, though they all ideally draw from some real-world artistic techniques under the assumption that such techniques developed by humans have inherent value due to the evolutionary nature of art. In the existing NPR literature, the commercial illustrative styles which inspired the look of *Team Fortress 2* are most closely related to the technical illustration techniques codified in [GGS+98]. In Gooch shading, the traditional Phong model [Phong75] is modified using a cool-to-warm hue shift to indicate surface orientation relative to a given light source. As a result, extreme lights and darks are reserved for edge lines and highlights, resulting in a clearer perception of 3D object structure under difficult lighting situations than traditional computer graphics lighting models. In the world of *Team Fortress 2*, characters and other objects can be viewed under a wide variety of lighting conditions and thus we employ a similar system so that characters are clearly identifiable and aesthetically pleasing even in difficult lighting situations.

While the Gooch shading algorithm maps an unclamped Lambertian term to a warm-to-cool hue shift to improve shape perception, others have created a cel-shaded look by mapping this term to a very small set of colors with a hard transition at the terminator (where the Lambertian term crosses zero) [Decaudin96] [LMH+00] [BTM06]. To achieve a cel-shaded look, Decaudin rendered objects with constant diffuse colors and relied upon shadow mapping to attenuate pixels facing away from a given light source. Lake does not rely upon shadow mapping but instead uses a 1D texture lookup based upon the Lambertian term to simulate the limited color palette cartoonists use for painting cels. Lake's technique also allows for the inclusion of a view-independent pseudo specular highlight by including a small number of bright texels at the "lit" end of the 1D texture map. Most recently, Barla has extended this technique by using a 2D texture lookup to incorporate view-dependent and level-of-detail effects. Barla also uses a Fresnel-like term to create a hard "virtual backlight" which is essentially a rim-lighting term, though this term is not designed to correspond to any particular lighting environment.

3.4 Commercial Illustration Techniques

In the work of the early 20th century commercial illustrators J. C. Leyendecker, Dean Cornwell and Norman Rockwell as well as our own internal concept art, we observed the following consistencies which we used to define the look of *Team Fortress 2*:

- Shading obeys a warm-to-cool hue shift. Shadows go to cool, not black
- Saturation increases at the terminator with respect to a given light source. The terminator is often reddened.
- High frequency detail is omitted where possible
- On characters, interior details such as clothing folds are chosen to echo silhouette shapes
- Silhouettes are emphasized with rim highlights rather than dark outlines

With these fundamental principles in mind, we set out to create art assets (characters, environments and texture maps) and real-time shading algorithms that result in renderings with these properties. In the next section, we will discuss creation of art assets for *Team Fortress 2*, before moving on to the technical shading details in Section 3.6.

3.5 Creating Art Assets

In this section, we will discuss 3D character and world modeling as well as the principles we followed when generating texture maps necessary to meet both our gameplay and artistic goals.

3.5.1 Character Modeling

Players of multiplayer combat games such as *Team Fortress 2* must be able to visually identify other players very quickly at a variety of distances and viewpoints in order to assess the possible threat. In *Team Fortress 2* in particular, the player's class—Demo, Engineer, Heavy, Medic, Pyro, Spy, Sniper, Soldier or Scout—is extremely important to gameplay and hence the silhouettes of the nine classes were carefully designed to be very distinct from one another, as shown in Figure 2.

The body proportions, weapons and silhouette lines as determined by footwear, hats and clothing folds were explicitly designed to give each character a unique silhouette. In the shaded interior areas of a character, the clothing folds were explicitly designed to echo silhouette shapes in order to emphasize silhouettes, as observed in the commercial illustrations which inspired our designs. As we will discuss in Section 3.6, the shading algorithms used on these characters complement our modeling choices to enhance shape perception.

Figure 2*. The nine character classes of* Team Fortress 2 *were designed to be visually distinct from one another. Even when viewed only in silhouette with no internal shading at all, the characters are readily identifiable to players. While characters never appear in such unflattering lighting conditions in Team Fortress 2, demonstrating the ability to visually read the characters even with no internal detail was used to validate the character design during the concept phase of the game design.*

3.5.2 World Modeling

The unique look of the world of *Team Fortress 2* is borne out of well-defined design principles. For the architectural elements of the world associated with each of the two teams, blue and red, we defined specific contrasting properties. While the red team's base tends to use warm colors, natural materials and angular geometry, the blue team's base is composed of cool colors, industrial materials and orthogonal forms, as illustrated by the concept paintings of opposing building structures in the top row of Figure 3.

Ultimately, the geometry of the game environments was modeled on these concept paintings, as shown in the bottom row of Figure 3. Though there is clearly more detail in the 3D modeled world than there is in the concept paintings, we still we deliberately avoided modeling the world in an overly complex or geometrically off-kilter manner as this would add an unnecessary level of visual noise - not to mention memory-hungry vertices - to the scene. We also found that keeping repetitive structures such as the bridge trusses, telephone poles or railroad ties to a minimum is preferable for our style, as conveying the impression of repetition in the space is more important than representing every detail explicitly.

By maintaining a minimal level of repetition and visual noise, we serve many of our gameplay goals while employing an almost impressionistic approach to modeling. This philosophy was also central to our texture painting style throughout the game.

***Figure 3**. World concept art for blue and red team bases (top) and in-game screenshots from* Team Fortress 2 *(bottom).*

3.5.3 Texture Painting

In *Team Fortress 2*, colors used on characters and the game world border on realism, but with increased saturation and value contrast. The blue and red teams in the game each have one base in a game level. The red and blue colors used to paint opposing bases are analogous to one another, as guided by the reference color swatch in Figure 4, with muted colors dominating and small areas of saturation to give further visual interest.

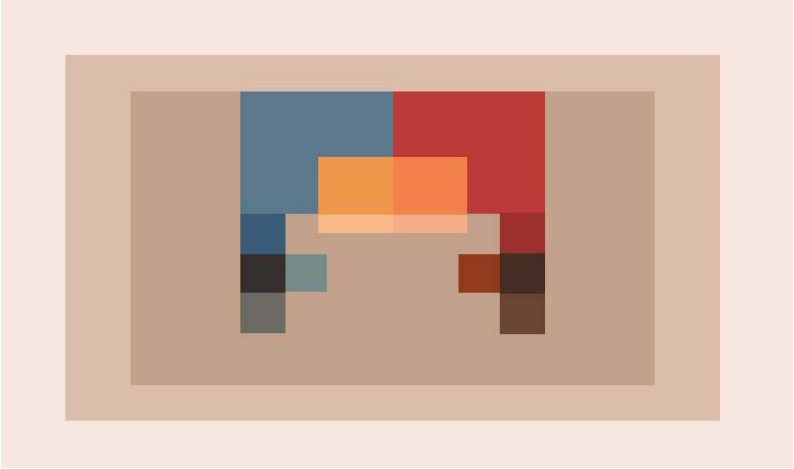


***Figure 4**. Color scheme for the opposing blue and red teams*

In addition to the dominant reds and blues, secondary and tertiary complimentary colors are added in smaller environmental props such as fire extinguishers and telephones. In general, the texture maps used on the 3D world are impressionistic, meaning that they are painterly and maintain a minimum level of visual noise. This is consistent with the style of painting used on background plates in many animated films, particularly those of Hayao Miyazaki, in which broad brush strokes appear in perspective, as if present in the 3D world rather than on the 2D image plane [Miyazaki02]. For our 3D game, we apply this same approach because we feel that its inherent frame-to-frame coherence has superior perceptual properties to an image-space painterly approach. Of course, it also helps that portraying brush strokes on the surfaces of 3D objects in a game world rather than the 2D image plane is already supported in any 3D game engine by definition.

2D Texture *Texture applied in 3D*

***Figure 4**. World texture with loose, visible brush strokes*

Much of the world texture detail in *Team Fortress 2* comes from hand-painted albedo textures which intentionally contain loose details with visible brush strokes that are intended to portray the tactile quality of a given surface, as shown in Figure 5. In the early stages of development, many of these 2D textures were physically painted on canvas with watercolors and scanned to make texture maps. As we refined the art style of the game, texture artists shifted to using photorealistic reference images with a series of filters and digital brush strokes applied to achieve the desired look of a physically painted texture.

Not only does this hand-painted source material create an illustrative NPR style in rendered images, but we have found that these abstract texture designs hold up under magnification better than textures created from photo reference due to their more intentional design and lack of photo artifacts. Furthermore, we believe that high frequency geometric and texture detail found in photorealistic games can often overpower the ability of designers to compose game environments and emphasize gameplay features visually using intentional design choices such as changes in color value.

Now that we have discussed *Team Fortress 2* asset creation, we will move on to the unique aspects of our character and model shading algorithms which work together with our asset choices to achieve a unique illustrative style and enable players to easily identify other players in the scene.

3.6 Interactive Character and Model Shading

In this section, we will discuss the non-photorealistic shading algorithms used on the characters and other models in *Team Fortress 2* in order to achieve our desired illustrative style. For characters and most other models in our game worlds, we combine a variety of view independent and view dependent terms as shown in Figure 7. The view independent terms consist of a spatially-varying directional ambient term plus modified Lambertian lighting terms. The view dependent terms are a combination of Phong highlights and customized rim lighting terms. All lighting terms are computed per pixel and most material properties, including normals, albedos, specular exponents and various masks are sampled from texture maps. In the following two sections, we will discuss how each of these lighting terms differs from conventional approaches and contributes to our aesthetic goals.

3.6.1 View Independent Lighting Terms

The view-independent lighting terms in *Team Fortress 2* consist of a summation of modified Lambertian terms and a spatially varying directional ambient term. The view-independent lighting terms can be summarized in Equation 1:

$$k_d \left[a(\hat{n}) + \sum_{i=1}^{L} c_i w\left(\left(\alpha(\hat{n} \cdot \hat{l}_i) + \beta \right)^{\gamma} \right) \right] \qquad (1)$$

where L is the number of lights, i is the light index, c_i is the color of light i, k_d is the albedo of the object sampled from a texture map, $\hat{n} \cdot \hat{l}_i$ is a traditional unclamped Lambertian term with respect to light i, the scalar constants α, β and γ are a scale, bias and exponent applied to the Lambertian term, $a()$ is a function which evaluates a directional ambient term as a function of the per-pixel normal $\hat{n}$ and $w()$ is a warping function which maps a scalar in the range of 0..1 to an RGB color.

Half Lambert One unusual feature of Equation 1 is the scale, bias and exponentiation applied to $\hat{n} \cdot \hat{l}_i$. Since our first game *Half-Life*, which shipped in 1998, we have been applying a scale by 0.5, bias by 0.5 and square to diffuse lighting terms to prevent characters from losing a sense of shape on the back side with respect to a given light source (α = 0.5, β = 0.5 and γ = 2). Even in our games which feature a more photorealistic look, we perform this operation so that the dot product which normally lies in the range of -1 to +1, instead lies in the range of 0 to 1 and has a pleasing falloff [MMG06]. Due to the scale and bias of the Lambertian term by 0.5, we refer to this technique as “Half Lambert.” This kind of scale and bias of the traditional Lambertian term appears in other NPR shading work including [RBD06]. In *Team Fortress 2*, we leave a and b at 0.5 but set the exponent g to 1 since we can express any shaping that we might want to get from the exponentiation in the warping function $w()$.

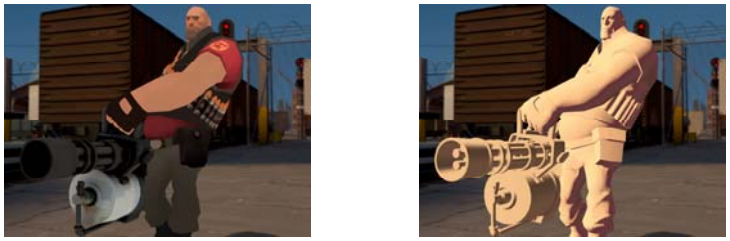
Diffuse Warping Function The second interesting feature of Equation 1 is the warping function $w()$ which is applied to the Half Lambert term. The goal of this function is to retain the shading information conveyed by the Half Lambert term while introducing the dramatic terminator observed in commercial illustration. In *Team Fortress 2*, this warping function is evaluated with a lookup into the artist-generated 1D texture shown in Figure 6. This is the same approach taken by [LCD06] but instead of using this texture indirection to create a cartoon "hard shading" look, we preserve the variation in illumination for all normals while tightening the transition from light to dark around the terminator as shown in Figure 7b.

Besides the general "shaping" of the lighting described above, the 1D light warping texture in Figure 6 has a number of interesting features. First, the rightmost value in the texture is not white but is, rather, only slightly brighter than mid-gray. This is done because there is a multiplication by 2 in the pixel shader after the lookup from this texture map. This allows the artists to paint values into this 1D lookup texture which are up to two times "overbright," meaning that while the input to this function is a Half Lambert term in the 0..1 range, the output is in the 0..2 range. It is also important to note that this texture has essentially three regions: a grayscale gradient on the right, a cool gradient on the left and a small reddish terminator region in the middle. This is consistent with our observations that illustrated shadows often tend toward cool colors, not black, and that there is often a slight reddening at the terminator. We will discuss potential future extensions to this model such as tuning the warping function to suit the hue of the underlying albedo in Section 3.8. As shown in Equation 1, this warping function is applied to the scalar Half Lambert term for each of the diffuse light sources affecting an object, resulting in an RGB color which is subsequently modulated with c_i, the color of the light source, resulting in a diffuse lighting component as illustrated in Figure 7b.

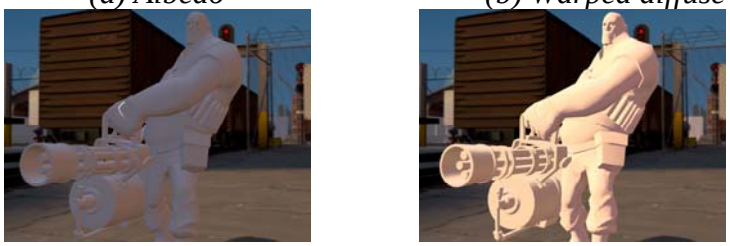
Directional Ambient Term In addition to the simple summation of warped diffuse lighting terms, we also apply a directional ambient term, $a(\hat{n})$. Though the representation is different, our directional ambient term is equivalent to an irradiance environment map, as discussed in [RH01]. Rather than a 9-term spherical harmonic basis, however, we use a novel 6-term basis which we call an "ambient cube," using cosine-squared lobes along positive and negative *x*, *y* and *z* axes [McTaggart04] [MMG06]. These ambient cubes are precomputed by our offline radiosity solver and stored in an irradiance volume for fast access at run time [GSH+98]. Despite the simplicity of this lighting component, shown in isolation in Figure 7c, the ambient cube term contributes bounced light which is critical to truly grounding characters and other models in the game world. The summation of these view-independent lighting terms, shown in 6d, is multiplied with k_d, the albedo of the base material (6a), resulting in a diffusely lit character as shown in Figure 7e.

Now that we have discussed our modifications to traditional view-independent lighting algorithms, we will move on to the extensions we have made to typical approaches to view-dependent lighting.

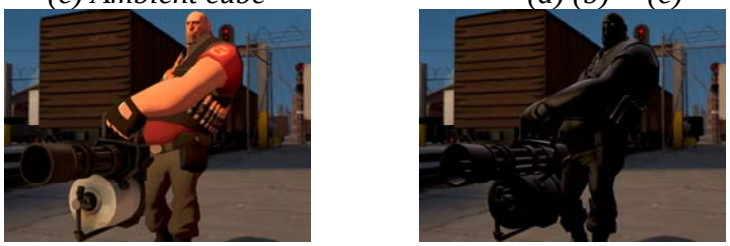
Figure 6*. Typical diffuse light warping function*



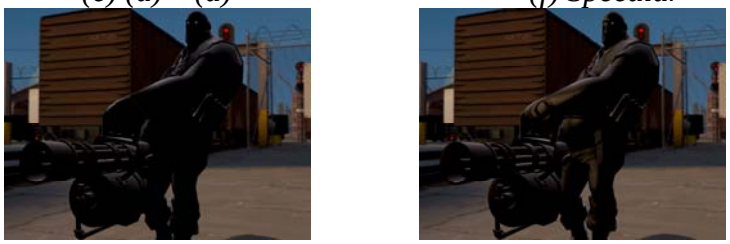
(a) Albedo *(b) Warped diffuse*



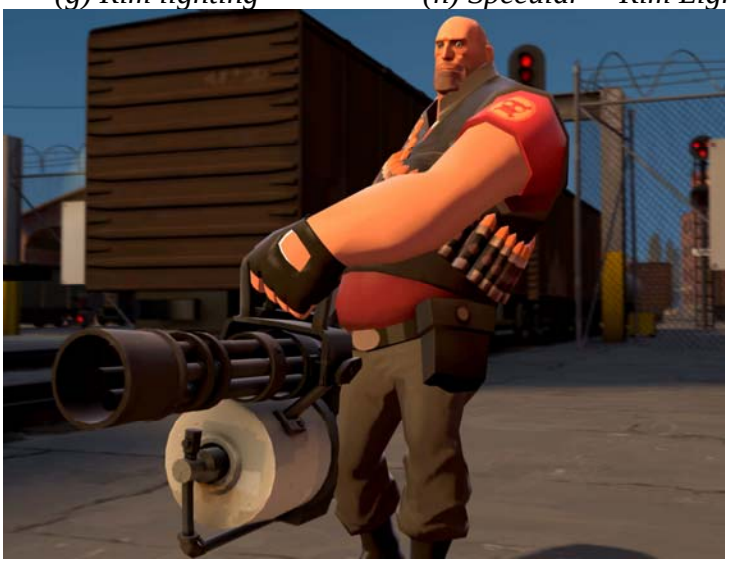
(c) Ambient cube *(d) (b) + (c)*



*(e) (a) * (d)* *(f) Specular*



(g) Rim lighting *(h) Specular + Rim Lighting*



(i) Final Result

***Figure 7**. Individual character and model shading terms*

3.6.2 View Dependent Lighting Terms

Our view dependent lighting terms consist of traditional Phong highlights combined with a set of customized rim lighting terms as summarized in Equation 2.

$$\sum_{i=1}^{L}\left[c_i k_s \max\left(f_s(\hat{v}\cdot\hat{r}_i)^{k_{spec}},\, f_r k_r(\hat{v}\cdot\hat{r}_i)^{k_{rim}}\right)\right]+(\hat{n}\cdot\hat{u})f_r k_r a(\hat{v}) \qquad (2)$$

where L is the number of lights, i is the light index, c_i is the color of light i, k_s is a specular mask painted into a texture channel, $\hat{v}$ is the view vector, f_s is an artist-tuned Fresnel term for general specular highlights, $\hat{r}_i$ is the reflection of the light vector from light i about $\hat{n}$, u is a world-space up vector, k_{spec} is the specular exponent fetched from a texture map, k_{rim} is a constant exponent which controls the breadth of the rim highlights, f_r is another Fresnel term used to mask rim highlights (typically just $(1-(\hat{n}\cdot\hat{v}))^4$, k_r is a rim mask texture used to attenuate the contribution of the rim terms on certain parts of a model and $a(\hat{v})$ is an evaluation of the ambient cube using a ray from the eye *through* the pixel being rendered.

Multiple Phong Terms The left side of Equation 2 contains a summation of Phong highlights calculated with the familiar expression $(\hat{v}\cdot\hat{r}_i)^{k_{spec}}$ which is modulated with appropriate constants and a Fresnel term. However, inside the summation, we also combine each Phong highlight using a *max()* function with additional Phong lobes that use a different exponent k_{rim}, Fresnel term f_r and mask k_r. In *Team Fortress 2*, k_{rim} is constant for a whole object and significantly lower than k_{spec}, yielding broad rim highlights from the lights in the scene, independent of the object's material properties. These broad rim highlights are masked with a Fresnel term f_r, ensuring that they are only present at grazing angles (the very definition of a rim light). This combination of Phong highlights that match the material properties of a given object with broad rim highlights helps to give *Team Fortress 2* its signature illustrative look.

Dedicated Rim Lighting In situations where a character has moved away from the light sources in the game level, rim lighting based solely on Phong terms from local light sources may not be as prominent as we would like. For this reason, we also add in the dedicated rim lighting term shown on the right side of Equation 2. This term consists of an evaluation of the ambient cube using the vector from the eye through the point being shaded $a(\hat{v})$ modulated with an artist-painted mask texture k_r, Fresnel term f_r and the expression $(\hat{n}\cdot\hat{u})$. This last expression is merely the per-pixel normal dotted with the up vector, clamped to be positive. This causes the dedicated rim lighting term to appear to add in indirect light from the environment, but only for upward facing normals. This is both an aesthetic choice and a perceptual decision designed exploit the human instinct to assume that lighting tends to come from above. Unlike photorealistic games, which place a major emphasis on micro details for realism, we feel that our impressionistic approach favors the audience of fast-paced action games who typically play a game like *Team Fortress 2* many thousands of times and are more concerned with perceiving gross shape and shading as emphasized by our focus on distinctive silhouettes and rim lighting.

The complete pixel shader used on characters and other models in *Team Fortress 2* is merely the summation of Equations 1 and 2 in addition to some other operations such as optional environment mapping and a distance fog term, which we have left out for brevity. Since we evaluate fog directly in pixel shader code, we chose not to build it into the per-light texture indirection as proposed by [BTM06].

3.7 Abstraction

Abstraction at distance is an important property of many NPR systems. For the most part, we rely on automatic mechanisms of abstraction such as depth fogging, minification of normal maps which tends to smooth out diffuse lighting, as well as SpecVar mapping to attenuate and broaden specular highlights at a distance [Conran05]. Since we do not apply image-space techniques such as outlining, we do not have to deal with issues of varying line weights, though our designers do intentionally simplify the shape and shading of our 3D skybox. In our graphics engine, the 3D skybox is a separate model which surrounds the game world but which is unreachable by players. This distant environment is not merely a painted backdrop on the inside of some simple geometry such as a large cube or sphere, but is a distant geometric model which provides parallax cues within itself and relative to the reachable game world. For *Team Fortress 2* in particular, 3D skybox geometry tends to be more painterly and is specifically modeled with less detail than it would be if it were in the interactive portions of the environment. This is not just to manage level of detail, but also fits with the overall visual style and prevents the skybox from generating high-frequency noise that would distract players.

3.8 Future Work

In future projects that call for illustrative rendering, we would like to further extend the model described above. For example, we have already experimented with extending our modification of the traditional Lambertian term to increase saturation of the particular hue of the albedo texture. To do this, we compute the hue of the albedo using shader operations and use the hue as the second coordinate into a 2D map which is an extension to the 1D map shown in Figure 7, where hue varies along the new axis. In practice, even a tightly optimized RGB-to-Hue conversion routine compiles to more than twenty pixel shader cycles, and we weren't willing to bear this expense on *Team Fortress 2* era hardware. In the future, we would like to be able to include this kind of extension to our diffuse model.

For our specular model, we have employed traditional Phong highlights in addition to our dedicated rim lighting. At the very least, we would like to extend this to allow for anisotropic materials such as cloth or brushed metals, using a combination of methods from [GSG+99], [HS98] and [GSG+99]. It might also be interesting to use stylized material-specific highlights by employing translation, rotation, splitting and squaring operations as discussed in [AH03].

On parts of models which have relatively coarse tessellation, the slowly-varying tangent frames can result in overly-broad Fresnel terms, leading to rim highlights that are more

obtrusive than we would like. In the future, we would like to look for methods to address this so that we can reliably generate more consistent and subtle rim highlights on polygonal models of varying mesh density.

Since our Source game engine has been previously used to develop a set of games such as the *Half-Life 2* series, *Counter-Strike: Source* and *Day of Defeat: Source*, which employ a more photorealistic rendering style, we have access to existing techniques for generating realistic effects such as motion blur, high dynamic range rendering, environment mapping as well as reflective and refractive water [MG06]. While rendering water with faithful photorealistic reflections of an otherwise NPR world is compelling, we would like to experiment with processing our reflection, refraction and environment maps with a filter such as an edge-preserving median filter to further stylize our water and other reflective effects.

Many traditional artists use image-space lightening and darkening techniques to increase contrast at important feature edges, as discussed in [LCD06]. We believe it would be appropriate for our visual style to use scene depth information to integrate this type of technique into the look of *Team Fortress 2*.

3.9 References

[AH03] ANJYO, K., AND HIRAMITSU, K. 2003. Stylized Highlights for Cartoon Rendering and Animation. IEEE Comput. Graph. Appl. 23, 4, 54–61.

[BTM06] BARLA, P., THOLLOT, J., AND MARKOSIAN, L. 2006. X-Toon: An Extended Toon Shader. In International Symposium on Non-Photorealistic Animation and Rendering (NPAR), ACM.

[CONRAN05] CONRAN, P. 2005. SpecVar Maps: Baking Bump Maps into Specular Response. In SIGGRAPH '05: ACM SIGGRAPH 2005 Sketches, ACM Press, New York, NY, USA, 22.

[DECAUDIN96] DECAUDIN, P. 1996. Cartoon Looking Rendering of 3D Scenes. Research Report 2919, INRIA, June.

[GGS+98] GOOCH, A. A., GOOCH, B., SHIRLEY, P., AND COHEN, E. 1998. A Non-Photorealistic Lighting Model for Automatic Technical Illustration. ACM Press/ACM SIGGRAPH, New York, M. Cohen, Ed., 447–452.

[GSG+99] GOOCH, B., SLOAN, P.-P. J., GOOCH, A. A., S SHIRLEY, P., AND RIESENFELD, R. 1999. Interactive Technical Illustration. In Proceedings of the 1999 Symposium on Interactive 3D Graphics, ACM Press, New York, J. Rossignac, J. Hodgins, and J. D. Foley, Eds., 31–38.

[GSH+98] GREGER, G., SHIRLEY, P., HUBBARD, P. M., AND GREENBERG, D. P. 1998. The Irradiance Volume. IEEE Computer Graphics and Applications 18, 2 (/), 32–43.

[HS98] HEIDRICH, W., AND SEIDEL, H. 1998. Efficient Rendering of Anisotropic Surfaces Using Computer Graphics Hardware. In Image and Multi-dimensional Digital Signal Processing Workshop, 315–318.

[LMH+00] LAKE, A., MARSHALL, C., HARRIS, M., AND BLACKSTEIN, M. 2000. Stylized Rendering Techniques for Scalable Real-Time 3D Animation. ACM Press, New York, J.-D. Fekete and D. Salesin, Eds., 13–20.

[LCD06] LUFT, T., COLDITZ, C., AND DEUSSEN, O. 2006. Image Enhancement by Unsharp Masking the Depth Buffer. ACM Transactions on Graphics 25, 3 (jul), 1206–1213.

[MG06] MCTAGGART, G., AND GREEN, C. 2006. High Dynamic Range Rendering in Valve's Source Engine. In ACM SIGGRAPH 2006 Course Notes. Course on High-Dynamic-Range Imaging: Theory and Applications. ACM Press/ACM SIGGRAPH.

[MCTAGGART04] MCTAGGART, G. 2004. Half-Life 2 Shading. In Game Developers Conference. Direct3D Tutorial.

[MMG06] MITCHELL, J. L., MCTAGGART, G., AND GREEN, C. 2006. Shading in Valve's Source Engine. In SIGGRAPH Course on Advanced Real-Time Rendering in 3D Graphics and Games.

[MIYAZAKI02] MIYAZAKI, H. 2002. The Art of Spirited Away. VIZ Media.

[PHONG75] PHONG, B. T. 1975. Illumination for Computer Generated Pictures. Commun. ACM 18, 6, 311–317.

[RH01] RAMAMOORTHI, R., AND HANRAHAN, P. 2001. An Efficient Representation for Irradiance Environment Maps. In SIGGRAPH 2001: Proceedings of the 28th annual conference on Computer graphics and interactive techniques, 497–500.

[RBD06] RUSINKIEWICZ, S., BURNS, M., AND DECARLO, D. 2006. Exaggerated Shading for Depicting Shape and Detail. SIGGRAPH 2006 25, 3 (July), 1199–1205.

[SCHAU74] SCHAU, M. 1974. J. C. Leyendecker. Watson-Guptill.

Chapter 4

Animated Wrinkle Maps

Christopher Oat[6]

AMD

Figure 1. *Ruby wrinkles her brow and purses her lips to express disapproval in the real-time short "Ruby: Whiteout".*

4.1 Abstract

An efficient method for rendering animated wrinkles on a human face is presented. This method allows an animator to independently blend multiple wrinkle maps across multiple

[6] email: chris.oat@amd.com

regions of a textured mesh such as the female character shown in Figure 1. This method is both efficient in terms of computation as well as storage costs and is easily implemented in a real-time application using modern programmable graphics processors.

4.2 Introduction

Compelling facial animation is an extremely important and challenging aspect of computer graphics. Both games and animated feature films rely on convincing characters to help tell a story and an important part of character animation is the character's ability to use facial expression. Without even realizing it, we often depend on the subtleties of facial expression to give us important contextual cues about what someone is saying to us. For example a wrinkled brow can indicate surprise while a furrowed brow may indicate confusion or inquisitiveness.

In order to allow artists to create realistic, compelling characters we must allow them to harness the power of subtle facial expression. The remainder of these notes will describe a technique for artist controllable wrinkles. This technique involves compositing multiple wrinkle maps using a system of masks and artist animated weights to create a final wrinkled normal map that is used to render a human face.

4.3 Wrinkle Maps

Wrinkle maps are really just bump maps that get added on top of a base normal map. Figure 2 illustrates a typical set of texture maps used for a female face: an albedo map, normal map, and two wrinkle maps. The normal map and the wrinkle maps store surface normals in tangent space [Blinn78]. The first wrinkle map encodes wrinkles for a stretched expression (exaggerated surprise expression: eyes wide open, eyebrows up, forehead wrinkled, mouth open) and the second wrinkle map encodes wrinkles for a compressed expression (think of sucking on a sour lemon: eyes squinting, forehead compressed down towards eyebrows, lips puckered, chin compressed and dimpled).

Figure 2. *Ruby's face textures (left to right): albedo map, tangent space normal map, tangent space wrinkle map 1 for stretched face, and tangent space wrinkle map 2 for compressed face.*

As usual, the normal map encodes fine surface detail such as pores, scars, or other facial details. Thus the normal map acts as a base layer that is added to a given wrinkle map. Because the normal map includes important detail, we don't want to simply average the normal map with the wrinkle maps or some of surface details may be lost. Listing 1 includes HLSL shader code for adding a wrinkle map with a normal map in a way that preserves the details of both maps.

```
// Sample the normal map and the wrinkle map (both are in tangent space)
// Scale and bias to get the vectors into the [-1, 1] range
float3 vNormalTS = tex2D( sNormalMapSampler, vUV ) * 2 - 1;
float3 vWrinkleTS = tex2D( sWrinkleMapSampler, vUV ) * 2 – 1;

// Add wrinkle to normal map
float3 vWrinkledNormal = normalize( float3( vWrinkleTS.xy + vNormalTS.xy,
                                            vWrinkleTS.z  * vNormalTS.z ) );
```

Listing 1. *An example HLSL implementation of adding a normal map and a wrinkle map (both are in tangent space). Both maps include important surface details that must be preserved when they are added together.*

4.4 Wrinkle Masks and Weights

In order to have independently controlled wrinkles on our character's face, we must divide her face into multiple regions. Each region is specified by a mask that is stored in a texture map. Because a mask can be stored in a single color channel of a texture, we are able to store up to four masks in a single four channel texture as shown in Figure 3. Using masks allows us to store wrinkles for different parts of the face, such as chin wrinkles and forehead wrinkles, in the same wrinkle map and still maintain independent control over wrinkles on different regions of our character's face.

Each wrinkle mask is paired with an animated wrinkle weight. Wrinkle weights are scalar values and act as influences for blending in wrinkles from the two wrinkle maps. For example, the upper left brow (red channel of left most image in Figure 3) will have its own "Upper Left Brow" weight. Each weight is in the range [-1, 1] and corresponds to the following wrinkle map influences:

- **Weight == -1 :** *Full influence from wrinkle map 1 (surprised face wrinkles)*
- **Weight == 0 :** *No influence from either wrinkle map (just the base normal map)*
- **Weight == 1 :** *Full influence from wrinkle map 2 (puckered face wrinkles)*

A given weight smoothly interpolates between the two wrinkle maps; at either end of the range one of the wrinkle maps is at its full influence and at the center of the range (when the weight is zero) neither of the wrinkle maps has an influence on the underlying normal map. Listing 2 gives HLSL shader code that implements this weighting and blending scheme.

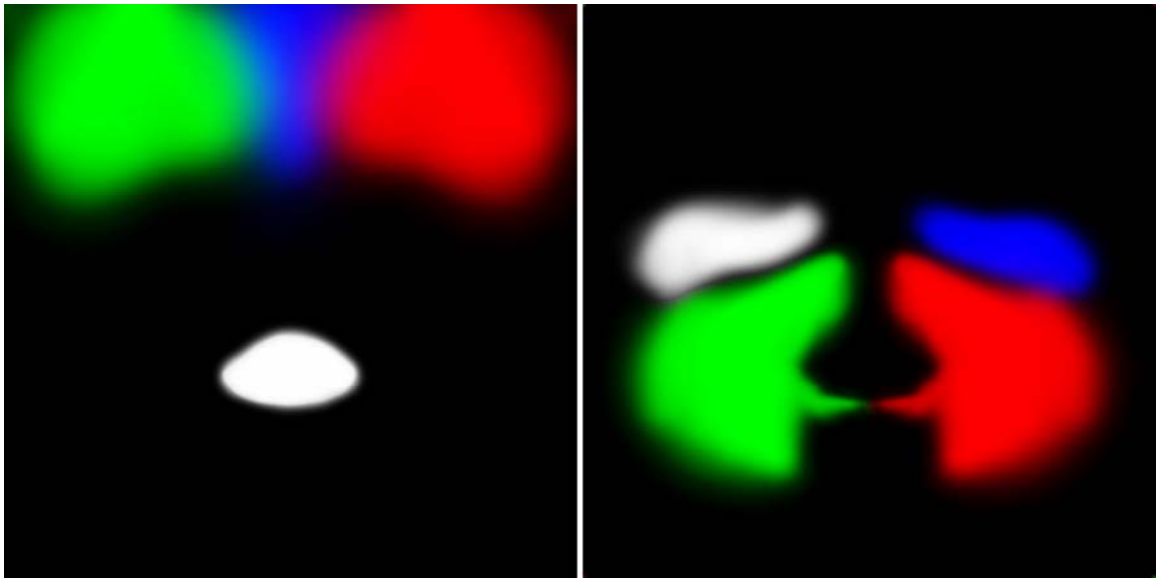


Figure 3. *Eight wrinkle masks distributed across the color and alpha channels of two textures (white represents the contents of the alpha channel). [Left] Masks for the left brow (red), right brow (green), middle brow (blue) and the lips (alpha). [Right] Masks for the left cheek (red), right cheek (green), upper left cheek (blue), and upper right cheek (alpha). We also use a chin mask which is not shown here.*

4.5 Conclusion

An efficient technique for achieving animated wrinkle maps has been presented. This technique uses two wrinkle maps (corresponding to squished and stretched expressions), a normal map, wrinkle masks, and artist animated wrinkle weights to independently control wrinkles on different regions of a facial mesh. When combined with traditional facial animation techniques such as matrix palate skinning and morphing this wrinkle technique can produce very compelling results that enable your characters to be more expressive.

4.6 References

[BLINN78] BLINN, J. Simulation of Wrinkled Surfaces, In Proceedings of ACM SIGGRAPH 1978, Vol. 12, No. 3, pp. 286-292, August 1978.

```
sampler2D sWrinkleMask1; // <LBrow, RBrow, MidBrow, Lips>
sampler2D sWrinkleMask2; // <LeftCheek, RightCheek, UpLeftCheek, UpRightCheek>

sampler2D sWrinkleMapSampler1; // Stretch wrinkles map sampler
sampler2D sWrinkleMapSampler2; // Compress wrinkles map sampler
sampler2D sNormalMapSampler;   // Normal map sampler

float4 vWrinkleMaskWeights1; // <LeftBrow, RightBrow, MidBrow, Lips>
float4 vWrinkleMaskWeights2; // <LCheek, RCheek, UpLeftCheek, UpRightCheek>

// Compute tangent space wrinkled normal
float4 ComputeWrinkledNormal ( float2 vUV )
{
   // Sample the mask textures
   float4 vMask1 = tex2D( sWrinkleMask1, vUV );
   float4 vMask2 = tex2D( sWrinkleMask2, vUV );

   // Mask the weights to get each wrinkle map's influence
   float fInfluence1 = dot(vMask1, max(0, -vWrinkleMaskWeights1)) +
                       dot(vMask2, max(0, -vWrinkleMaskWeights2));

   float fInfluence2 = dot(vMask1, max(0, vWrinkleMaskWeights1)) +
                       dot(vMask2, max(0, vWrinkleMaskWeights2));

   // Clamp the influence [0,1].  This is only necessary if
   // there are overlapping mask regions.
   fInfluence1 = min(fInfluence1, 1);
   fInfluence2 = min(fInfluence2, 1);

   // Sample the normal & wrinkle maps (we could branch here
   // if both influences are zero).  Scale and bias to get
   // vectors into [-1, 1] range.
   float3 vNormalTS = tex2D(sNormalMapSampler, vUV)*2-1;  // Normal map
   float3 vWrink1TS = tex2D(sWrinkleMapSampler1, vUV)*2-1;// Wrinkle map 1
   float3 vWrink2TS = tex2D(sWrinkleMapSampler2, vUV)*2-1;// Wrinkle map 2

   // Composite the weighted wrinkle maps to get a final wrinkle
   float3 vWrinkleTS;
   vWrinkleTS = lerp( float3(0,0,1), vWrink1TS, fInfluence1 );
   vWrinkleTS = lerp( vWrinkleTS, vWrink2TS, fInfluence2 );

   // Add final wrinkle to the base normal map
   vNormalTS = normalize( float3( vWrinkleTS.xy + vNormalTS.xy,
                                  vNormalTS.z * vWrinkleTS.z ) );

   return vNormalTS;
}
```

Listing 2. *An example HLSL function that takes a UV texture coordinate as an argument and returns a wrinkled normal that may be used for shading.*

Chapter 5

Terrain Rendering in Frostbite Using Procedural Shader Splatting

Johan Andersson [7]

DICE

5.1 Introduction

Many modern games take place in outdoor environments. While there has been much research into geometrical LOD solutions, the texturing and shading solutions used in real-time applications is usually quite basic and non-flexible, which often result in lack of detail either up close, in a distance, or at high angles.

One of the more common approaches for terrain texturing is to combine low-resolution unique color maps (Figure 1) for low-frequency details with multiple tiled detail maps for high-frequency details that are blended in at certain distance to the camera. This gives artists good control over the lower frequencies as they can paint or generate the color maps however they want.

For the detail mapping there are multiple methods that can be used. In *Battlefield 2*, a 256 m^2 patch of the terrain could have up to six different tiling detail maps that were blended together using one or two three-component unique detail mask textures (Figure 4) that controlled the visibility of the individual detail maps. Artists would paint or generate the detail masks just as for the color map.

Figure 1*. Terrain color map from Battlefield 2*

[7] email: johan.andersson@dice.se

***Figure 2**. Overhead view of Battlefield: Bad Company landscape*

***Figure 3**. Close up view of Battlefield: Bad Company landscape*

There are a couple of potential problems with all these traditional terrain texturing and rendering methods going forward, that we wanted to try to solve or improve on when developing our *Frostbite* engine.

Our main problem is that they are static. We have wanted to be able to destroy the terrain ever since *Battlefield 1942*, both geometrically and texture-wise, but haven't had the performance or memory to support arbitrary geometric destruction of the heightfields. Extending the texture compositing method for destruction by dynamically changing the compressed color maps and detail mask textures is also not really feasible. Neither is adding even more simultaneous detail map variations for destroyed materials such as cracked tarmac or burnt grass.

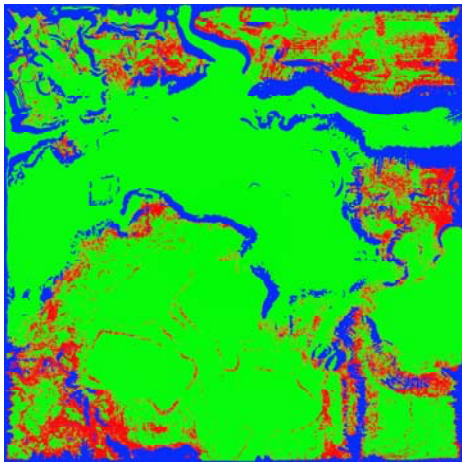


Figure 4*. Terrain detail mask texture from Battlefield 2. RGB channel intensity represents visibility of 3 separate tiling detail textures.*

At the same time as we wanted to be able to destroy the terrain, we also wanted to increase the visual quality of the terrain in general while reducing the memory usage. Traditional terrain texture compositing schemes such as the *Battlefield 2* unique color maps and detail mask textures takes a lot of memory and is a fixed feature and memory cost. It can be difficult and computationally prohibitive to vary the shading and compositing on different areas and materials on the terrain.

But varying and specializing shading and texturing for different materials is a very good thing to do and is usually the way shaders for ordinary meshes in games are done to be as memory and computationally efficient as possible.

For example: if we want to use parallax occlusion mapping ([Tatarchuk06]) on a rocky surface we do not want to pay the performance cost of computing parallax occlusion mapping for all other materials that do not need it. Same applies if we have a sea floor material that covers large parts of the level but the shading and texturing quality is not that important because it will be partially obscured by the sea surface. In that case we would like to not have to pay the cost of storing color maps and detail mask textures when the material could be approximated with a few tinted detail maps.

Specializing terrain shaders to different terrain materials opens up a lot of interesting possibilities and in this chapter we describe a terrain rendering system and technique built on that idea for DICE's Frostbite engine that is used in *Battlefield: Bad Company* for the Xbox 360 and PlayStation 3.

5.2 Terrain Texturing and Shading

The basic idea of the terrain texturing in Frostbite is to allow artists to create specialized shaders with arbitrary dynamic texture compositing for individual terrain materials and

distribute them over the terrain using the method most suited depending on the nature of the material.

5.2.1 Graph-based surface shaders

In order for artists to be able to easily experiment and create custom shaders for individual terrain materials we utilize an internal high-level shading framework that allows surface shaders to be authored trough a graph representation instead of code (Figure 5). See [AT07] for more details.

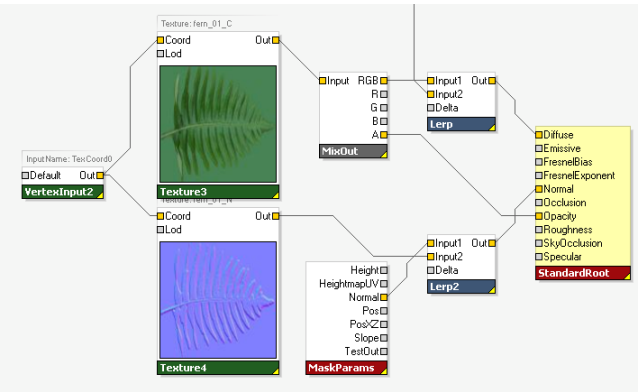


Figure5*. Example of graph-based surface shader*

There are multiple benefits with this graph-based approach for authoring terrain shaders:

- Artist-friendly. Very few of our artists know HLSL and tweaking values and colors trough standard dialogs instead of writing text is a big productivity gain.
- Flexibility. Both programmers and artists can easily expose and encapsulate new nodes and shading functionality.
- Data-centric. It is easy to automatically process or transform the data in the shading graph which can be very powerful and is difficult to do with code-based shaders.

The shading framework generates resulting shading solutions and the actual pixel and vertex shaders to use in-game via a complex but powerful offline processing pipeline. The framework generates the shaders based on the high-level rendering state combinations. A number of states are available to the system, such as the number and type of light sources, geometry processing methods, effects and surface shaders.

The pipeline-generated shading solutions are used by the game runtime which is implemented on multiple platforms through

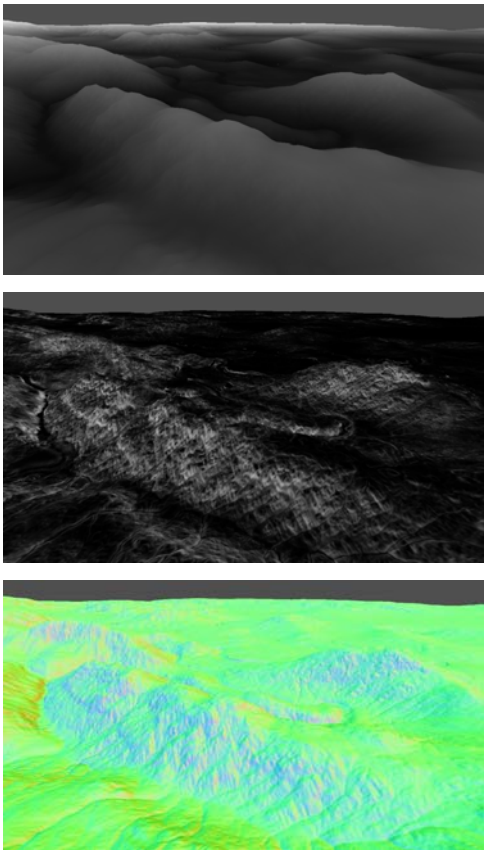


Figure6*. Terrain per-pixel parameters: height (top), slope (middle) and normal (bottom)*

rendering backends for DirectX9, Xbox 360, PlayStation 3 and Direct3D10. It handles dispatching commands to the GPU and can be quite thin by following the shading solutions that contain instructions on exactly how to setup the rendering.

Along with enabling graph-based shader development, we realized the need to support flexible and powerful code-based shader block development in our framework. Often, both artists and programmers may want to take advantage of custom complex functions and reuse them throughout the shader network. As a solution to this problem, we introduce *instance shaders* - shader graphs with explicit inputs and outputs that can be instanced and reused inside other shaders. Through this concept, we can hierarchically encapsulate parts of shaders and create very complex shader graph networks while still being manageable and efficient. This functionality allows the shader networks to be easily extensible.

Much of the terrain shading and texturing functionality is implemented with instance shaders. General data transformation and optimization capabilities in the pipeline that operate (mostly) on the graph-level are utilized to combine multiple techniques to create long single-pass shaders.

5.2.2 Procedural parameters

Over the last decade, the computational power of consumer GPUs has been exceeding the Moore's law, graphics chips becoming faster and faster with every generation. At the same time, memory size and bandwidth increases do not match the jumps in GPU compute. Realizing this trend, it makes sense to try to calculate much of the terrain shading and texture compositing in the shaders instead of storing it all in textures.

There are many interesting procedural techniques for terrain texturing and generation, but most would require multi-pass rendering into cached textures for real-time usage or can be expensive and difficult to mipmap correctly (such as GPU Wang Tiles in [Wei 03]).

We have chosen to start with a very simple concept of calculating and exposing three procedural parameters to the graph-based terrain shaders (Figure 6) for performance reasons:

- Height (meters)
- Slope (0.0 = 0 degrees, 1.0 = 90°)
- Normal (world-space)

Since the terrain is heightfield-based the parameters are simple and fast to compute for every pixel on the terrain.

The height is a bilinear sample of a normalized 16-bit heightfield texture and then scaled by the maximum height of the terrain.

The normal can be computed in multiple ways, we found that a simple four-sample cross filter works well enough for us (*Listing 1*).

The slope is one minus the *y*-component of the normal.

```
sampler bilinearSampler;
Texture2D heightmap;

float3 filterNormal(float2 uv, float texelSize, float texelAspect)
{
    float4 h;
    h[0] = heightmap.Sample(bilinearSampler, uv + texelSize*float2( 0,-1)).r * texelAspect;
    h[1] = heightmap.Sample(bilinearSampler, uv + texelSize*float2(-1, 0)).r * texelAspect;
    h[2] = heightmap.Sample(bilinearSampler, uv + texelSize*float2( 1, 0)).r * texelAspect;
    h[3] = heightmap.Sample(bilinearSampler, uv + texelSize*float2( 0, 1)).r * texelAspect;

    float3 n;
    n.z = h[0] - h[3];
    n.x = h[1] - h[2];
    n.y = 2;

    return normalize(n);
}
```

Listing 1*. Heightmap normal cross filter shader (Direct3D 10 HLSL)*

5.2.3 Masking

The terrain shaders determine how a material looks, but also, if wanted, where it appears.

For example, let's say we have a mountain material shader that we would like to be visible on the slopes of the terrain. This can be accomplished in two ways. One method is to use a grayscale mask texture can be manually painted (or generated in some other program) over the terrain giving full control where the material appears. Note that we would have to pay the price on memory cost for this mask's texture (since all the texture compositing is done at runtime).

The other method we support is to let the shader itself compute where it should appear. In this case for a mountain, a simple ramp function can be computed with the procedural slope parameter available in the shader to mask in the mountain between a specified min and max slopes together with a linear transition (Figure 7 and 8*)*. This method is also the base of many offline terrain rendering and generation programs such as [Terragen*].

The resolution of the mask computed from the procedural slope in the shader is limited by the resolution of the heightfields. Therefore at extreme close-ups the masks can become blurry due to bilinear texture magnification of the heightfields. This can create dull and unnaturally smooth transitions between materials. The same problem arises when using low-resolution image-based painted masks.

We can improve the bland transitions by adding detail on a per-material basis to the computed masks. We can add detail when necessary on a per-material basis to the computed masks by blending in tiled detail masks or procedural noise such as fractional Brownian motion.

Computing noise in pixel shaders can yield high quality and can be reasonably fast on modern GPUs ([Tatarchuk 07]) but for our purpose where we would like to compute multiple octaves for multiple materials it is still computationally prohibitive.

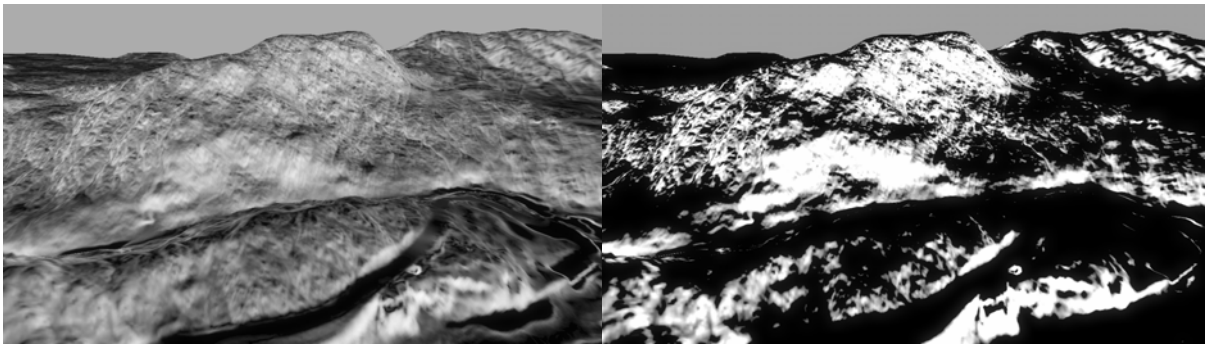


Figure7*. Terrain slope parameter (left). Mountain mask calculated in shader (right).*

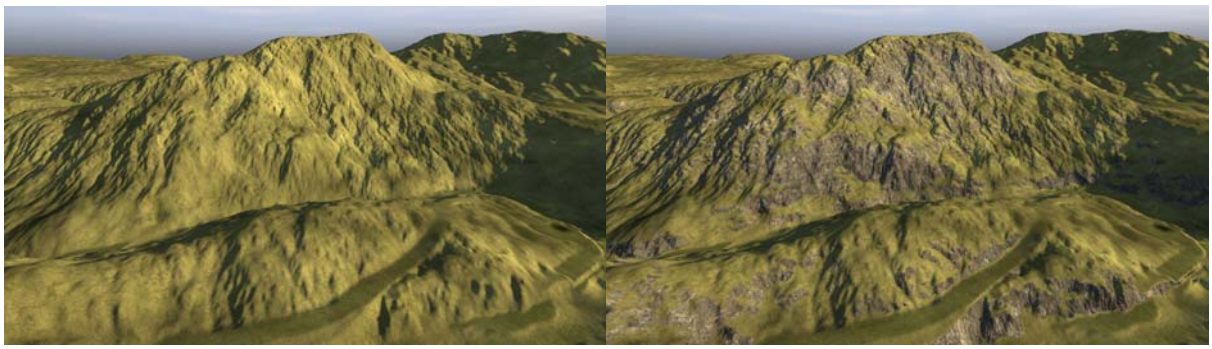


Figure8*. Terrain without (left) and with (right) mountain material that uses computed mask from the slope parameter*

fBm can also be "computed" in shaders by pre-generating a noise texture for a specific period offline and sample it for every octave instead of computing the noise function arithmetically. This is not as flexible and limits the range but can be faster and is still useful.

In our case, we increase mask detail for most of our materials with a more efficient and easy approach. We author or (in the shader) reuse tiled grayscale textures as detail mask textures and combine them with the lower resolution mask with various functions (Figure 9). This has the benefit of requiring few texture fetches (in contrast to the texture-based fBm method) and is flexible in ALU operation complexity (in contrast to ALU-based noise), and is therefore a good compromise. It also gives artists good control over the detail transitions by creating the detail mask textures and selecting how to combine the masks.

The Adobe® Photoshop™ blend mode *Overlay* (Listing 2) is very useful for combining two mask textures and adding detail. It does not affect areas where the base procedural mask is 0.0 or 1.0 so the base shape of the mask is kept. We use it almost exclusively together with simple multiplies and linear blends, but any blend mode can of course be used.

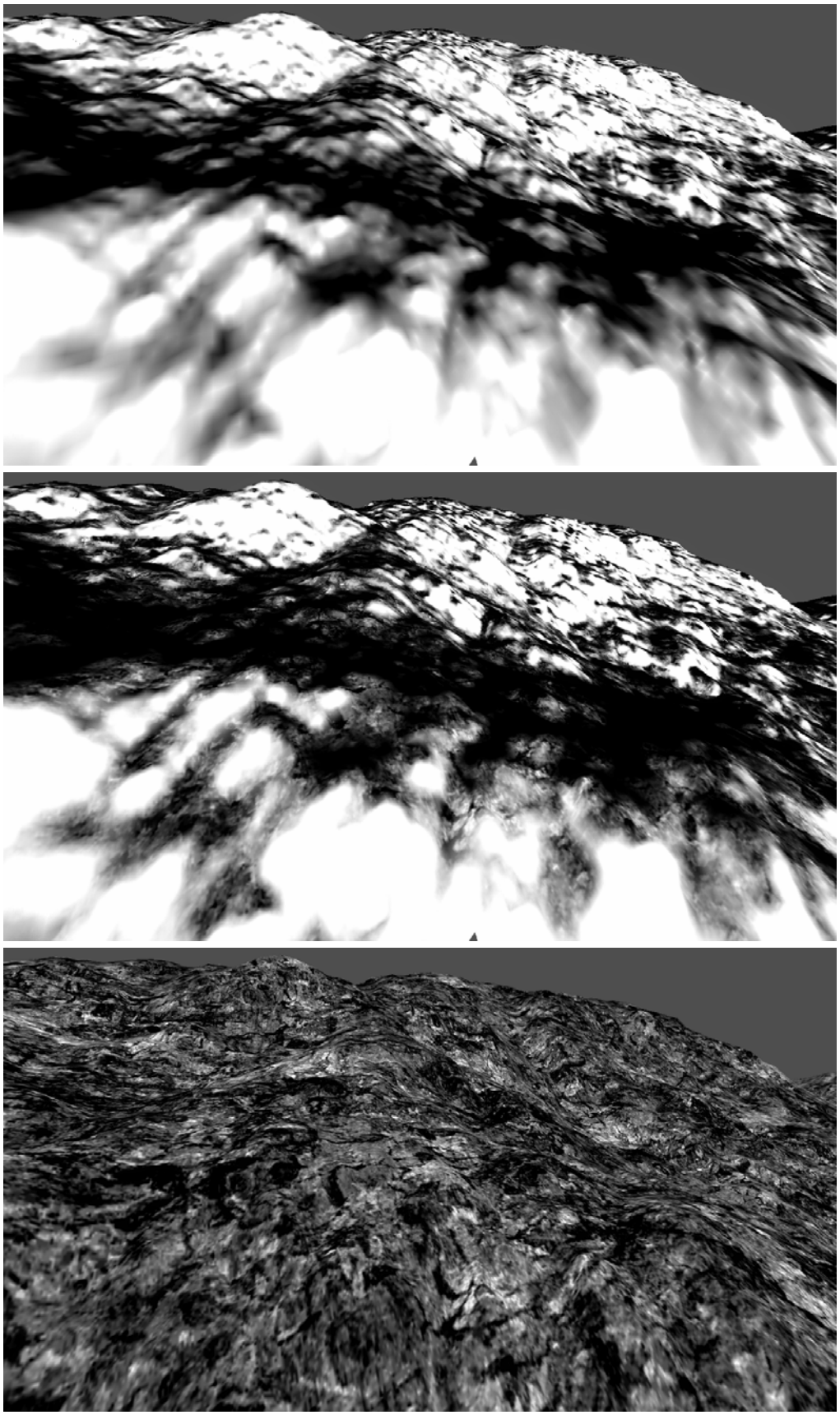


Figure9*. Close up of terrain with procedural slope mask (top). Procedural mask blended with tiled detail mask texture using overlay blend mode (middle). Tiled detail mask texture (bottom)*

Having multiple detail mask textures together with all other textures quickly eat performance and texture samplers (only 16 are available in Shader Model 3). To improve on this, we have had some good results with reusing channels of ordinary color textures, or even normal maps, already used in the shader, potentially with different tiling of the texture coordinates, and remapping the range or changing the contrast (*Listing 2*) to get a normalized mask value and use that instead of an extra texture.

```
float overlayBlend(float value1, float value2, float opacity)
{
    float blend = value1 < 0.5 ? 2*value1*value2 : 1 - 2*(1-value1)*(1-value2);
    return lerp(value1, blend, opacity);
}

float scaleContrast(float value, float contrast)
{
    return saturate((value-0.5)*contrast+0.5);
}
```

Listing 2*. Overlay blend and contrast HLSL functions. Works with values in normalized [0, 1] range and can be easily extended for arbitrary dimensions (colors for example).*

5.2.4 Static Sparse Mask Textures

There are many terrain materials that can not be generated in a purely procedural manner, especially when using only basic parameters, such as height, slope and normal. A good example are the open fields in a distance in Figure 1, they are artificially created and level designers and artists wanted full control of their shape and location.

To facilitate this, we support painting arbitrary grayscale masks over the terrain for individual terrain materials in our Editor tool or manually in Photoshop.

To save memory, all painted mask textures are stored in a sparse quad-tree texture representation that only stores unique 32×32 pixel tiles. This can be a big win since usually no terrain material mask covers the entire terrain (Figure 10) and those empty areas then do not take up any memory[1]. The quad-tree representation also allows areas in the mask texture that always will be viewed from a distance to be reduced in resolution.

For the best texture resource utilization and performance, four quad-tree mask textures are packed together into the R, G, B and A channels of one 64-bit indirection texture, one 32-bit quad-tree level texture and one DXT5A/BC4 atlas texture (Figure 11).

The indirection texture stores a normalized XZ index to the tile in the atlas to use.

The quad-tree level texture stores which level in the quad-tree the tile is on which is used when calculating the texture coordinates from world space positions.

In *Listing 3* the 4x sparse quad-tree mask texture sampling shader is included.

[1] Not entirely true, the indirection textures still take memory

***Figure10**. Source mask texture for leaf terrain material*

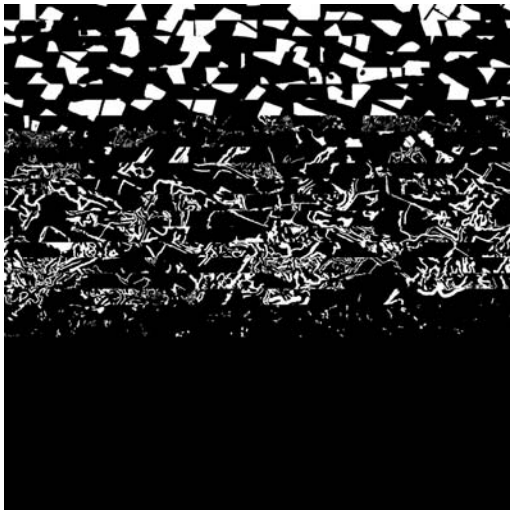


***Figure11**. 2048x2048 grayscale mask texture atlas with 32x32 tiles*

When creating and sampling from the mask texture atlas, care must be taken to pad the borders of the tiles to prevent filtering artifacts in the edges when bilinear filtering is used. Otherwise parts of the bordering tile in the atlas will leak over in the edges resulting in ugly line artifacts in the borders of the tiles.

In Direct3D10 and on Xbox 360, a texture array can be used instead of an atlas and then bilinear filtering automatically works correctly without any extra padding. Unfortunately texture arrays have a limit of 512 slices (tiles) in Direct3D10 and 64 slices on Xbox 360 which limits their usefulness in this case unless the tiles are split up over multiple texture arrays.

5.2.5 Destruction Mask

When an area of the terrain is affected by ground destruction, the pixels around that area in the heightfields are updated. We would like to, at the same time, be able to change the texture compositing and make other terrain materials visible for that specific area to show for example burnt dirt (Figure 12).

***Figure12**. Ground destruction masking in burnt dirt material on road*

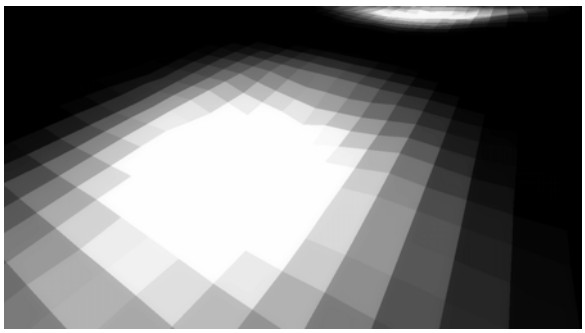


***Figure13**. Destruction mask (with point-filtering for clarity)*

```
sampler pointSampler;
sampler bilinearSampler;
Texture2D levelsTexture;
Texture2D indicesTexture;
Texture2D atlasTexture;

void sampleQuadTreeMasks(
    in float2 posXZ,     in float2 heightmapUV,
    in float2 atlasSize, in float2 invAtlasSize,
    in float maskIndirectionResolution,
    out float4 outMasks)
{
    float4 indices = indicesTexture.Sample(pointSampler, posXZ);
    float4 levels = levelsTexture.Sample(pointSampler, posXZ);

    levels *= 255.0f; // unpack [0,1] -> [0,255] 8-bit

    [unroll]
    for (int i=0; i<4; i++)
    {
        float2 uv = frac(heightmapUV*maskIndirectionResolution/levels[i]);
        uv *= invAtlasSize;

        float2 index;
        index.x = floor(fmod(indices[i], atlasSize.x));
        index.y = floor(indices[i] * invAtlasSize.x);

        uv += index*invAtlasSize;

        outMasks[i] = atlasTexture.Sample(bilinearSampler, uv);
    }
}
```

Listing 3*. Quad-tree texture sampling shader (Direct3D 10 HLSL), output is 4 individual mask values. Note: padding between tiles is not included.*

We do this by dynamically rendering textured mask decals into a unique destruction mask texture that covers the entire terrain that can be destructed (Figure 13). The destruction mask is very low resolution, 4 pixels per meter, because the mask only needs to contain rough circular gradients in the areas affected by ground destruction. More detail can be added in the shaders in a similar manner of adding detail to the procedural masks and the painted mask textures we described earlier in this chapter.

Nonetheless, even with a reasonably low resolution, texture memory footprint becomes a problem. In a 2048×2048 m destructible terrain area, a 4 pixel per meter uncompressed 8-bit mask texture takes $(2048 * 4)^2$ bytes = 64 MB. That is hardly desirable on any platform.

In our case, the worst case scenario for ground destruction is not really that 100% of the terrain area can be fully destroyed and need to be masked at the same time. The percentage we can get away with is much lower, perhaps 10%. But we do not want to restrict where on the terrain the destruction can happen, so the 10% destroyed area can be arbitrarily scattered over the entire terrain.

This is a similar scenario to static sparse mask textures that we encountered before, however with dynamic textures in this case. So what we chose to do to save memory is to create and incrementally update a dynamic sparse mask texture on the GPU for the ground destruction (Figure 14).

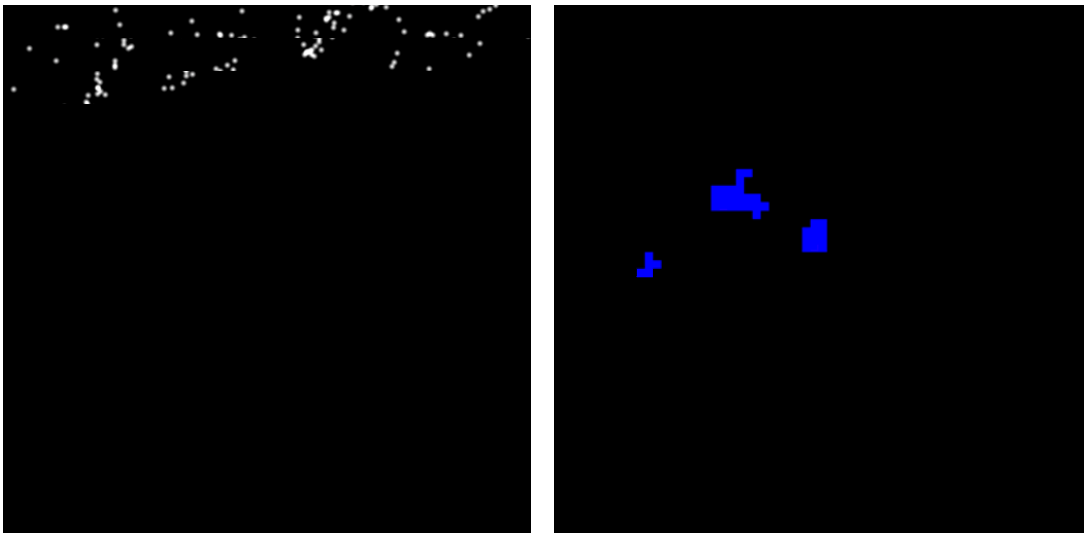


Figure14*. Dynamic sparse mask texture atlas render target (left). Dynamic mask indirection texture top-projected over terrain (right), B and G channels are normalized XY index into tile in atlas. Three independent areas on the terrain have been affected by ground destruction.*

We do not need to vary the resolution of the destruction mask, in contrast to the static mask textures, so we can use a fixed grid structure for the indirection texture and no quad-tree level texture making the texture sampling shader faster.

When a ground destruction event is triggered and the heightfield is updated, we check if the area the crater cover is allocated in the sparse texture. If it isn't, we allocate one or multiple new tiles in the atlas and store the XY index to the atlas tiles in a CPU-copy of the indirection texture. The indirection texture is then copied over to the GPU.

Each crater is represented as a small 2D texture-mapped decal that is rendered into the destruction mask texture atlas tiles by setting the viewport to match the tile and then rendering all decals within that tile. Since very few tiles are allocated or updated every frame, but the total amount of craters and allocated tiles can be high, this incremental update can be a big win.

5.3 Terrain Shader Compositing

Any area on the terrain can have multiple overlapping terrain materials that need to be composited together. The materials are specified in a strict order that determines which material lies on top of which.

A simple implementation to render the materials would be to do the compositing of the terrain materials in the frame buffer using alpha-blending a la [Bloom00]. Such an implementation would go through each terrain material in back-to-front order and render all terrain geometry associated with that material and blending its output on top of the previous material's output.

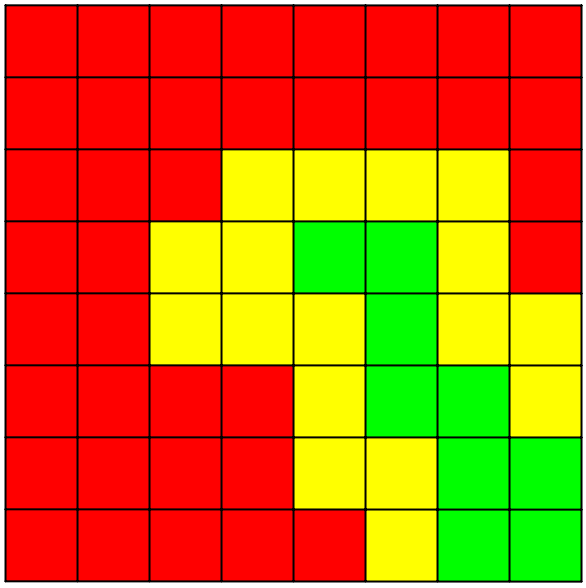


***Figure14**. 2 terrain materials (red and green) creating 3 terrain material combinations due to overlap (yellow)*

However there are quite a few drawbacks with such a multi-pass approach:

- Frame buffer bandwidth. The terrain covers much of the screen and the more materials we add the more times every pixel has to be read and written back to the frame buffer costing memory bandwidth.
- Geometry overdraw. As with any multi-pass technique, the geometry is rendered multiple times. Since we want to render the terrain with lots of triangles for good detail and GPU vertex throughput isn't increasing as fast as pixel throughput, this can become a big bottleneck.
- Duplicated shader computation. Many of the computations in the terrain material shaders such as the terrain normal would be recomputed for every pass which is costly.
- Fixed function blend modes. The built-in blend modes aren't very flexible, especially compared to shaders. There are lots of interesting methods to non-linearly combine natural textures for terrain

Instead, we combine all the terrain material shaders automatically into big single-pass shaders and do the compositing inside the shaders. This allows for optimal performance by sharing shader computations between materials while only rendering the geometry and pixels once.

To implement the compositing we have a pre-processing step which analyzes the terrain and gathers all terrain material combinations that are used on every patch of the terrain. This process takes into consideration the terrain material distribution masks and when multiple materials overlap on the same patch (Figure 14) all will be included.

The information is then used to create composite shaders for every terrain material combination found and a grid referencing the composite shaders is saved out so the runtime will know which shader to use for which patch and area on the terrain.

The composite shaders are surprisingly simple to create automatically given the graph-representation of the shaders. The outputs of the terrain material shaders are re-routed

to the inputs of a pre-created compositing shader that combines all the materials and outputs the final color.

Duplicate resources (textures, samplers and constants) and identical graph sub-trees or code in the composite shaders are automatically removed by the general shader graph compiler.

Figure15*. Overhead view of terrain with about 15 terrain material shaders masked & combined using Procedural Shader Splatting*

To improve performance further, dynamic flow control is used rigorously in the composite shaders to skip computations and texture fetches in areas that materials are fully covered by other materials. This is a big win on all platforms.

We call this method of terrain texturing and shading *procedural shader splatting*, from that we are arbitrarily "splatting" procedural shaders on various areas of the terrain and on top of each other and then combining them all for efficient rendering (Figure 15).

5.4 Terrain Rendering

The terrain culling and LOD is done via a frame-to-frame coherent quad-tree structure where every node knows the maximum and minimum height of the heightfield area within the node. The minimum height of a node may change when the heightfield is altered by ground destruction.

All visible leaf nodes in the quad-tree are rendered as fixed 33 x 33 vertex grids. The vertex grid is stored in a single shared vertex buffer and the grid vertices only contain a

4-byte UV coordinate that gives a [0,1] parameterization over the grid. This parameterization is transformed into both heightfield- and world-space in the vertex shader and used for fetching the terrain height for the vertex trough the heightfield texture. Because the vertex grid is aligned with the heightfield, point-filtering can be used which is a benefit on GPUs that does not natively support bilinear filtering of textures in vertex shaders (GeForce 6 and 7).

On platforms and graphics cards that do not efficiently support vertex texture fetch we have a pool of 33×33 vertices vertex buffers that are allocated on-demand on a LRU-basis to visible quad tree nodes and filled by CPU/SPU threads by sampling the heightfield.

The fixed vertex grid resolution is important to be able to support the worst case scenario with arbitrary ground destruction at a fixed cost and quality. This "wastes" triangles in non-altered flat areas but we found the cost to be worthwhile because of the simplicity and generality of this approach.

5.4.1 Geometry LOD

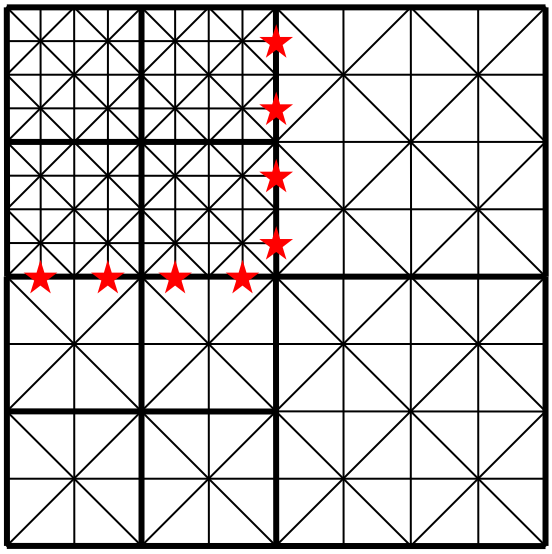


Figure16*. Quad tree patches with different LOD creating t-junctions and holes in the terrain at the red stars*

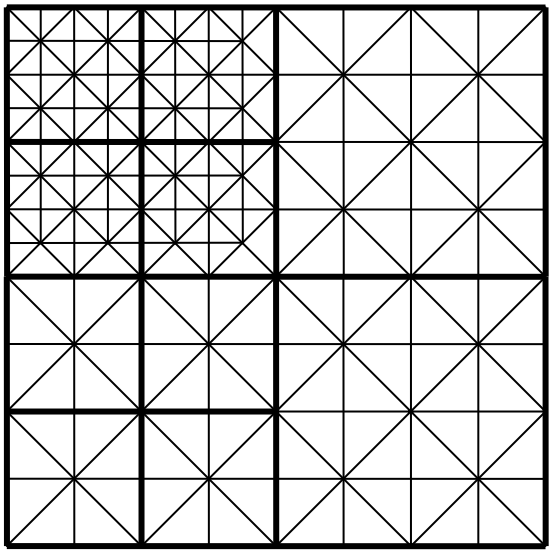


Figure17*. Triangles creating t-junctions removed in highest LOD patch*

As illustrated in Figure 16, neighboring quad tree patches of different LOD will create t-junctions in the more detailed patches when using a fixed grid resolution for the quad tree leaves.

If the terrain heightfield varies near the t-junction, the triangles of the t-junction in the detailed patch (high LOD) will not sample the same height in the heightfield as the triangle next to them in the lower LOD. This creates holes in the terrain that can be quite apparent, esp. if a bright color such as the sky is rendered below the terrain. To get rid of potential holes in the terrain we need to get rid of the t-junctions.

We can do this by first requiring that all quad tree nodes have a maximum of 1 level difference to its neighbors. Then replace the two triangles that make up every t-junction in the detailed patch with a single triangle that only uses vertices also available in the neighboring lower LOD (Figure 17). These vertices are guaranteed to exist by the max level difference we required.

To support all possible combinations of quad tree nodes with the max level difference restriction, all we need are 9 different index buffer permutations (Figure 18):

- One permutation that has all triangles in the vertex grid intact and is used when the neighbor patches are of the same level. This is the most common case.
- Four permutations with the t-junction triangles removed on one of the four sides of the patch
- Four permutations with the t-junction triangles removed on two sides next to each other
-

The reason why we do not need all possible sixteen permutations (t-junctions from all sides individually removed or kept) is that we chose to remove geometry from the detailed patches instead of adding geometry to the lower LOD patches (which also works). Two sides of a quad tree leaf node always shares the parent, and thus LOD, which means that we do not need to remove t-junctions from more than two sides next to each other of a patch.

This technique with multiple index buffer permutations is a bit similar to [Dallaire06], but working with quad-trees instead of same size patches.

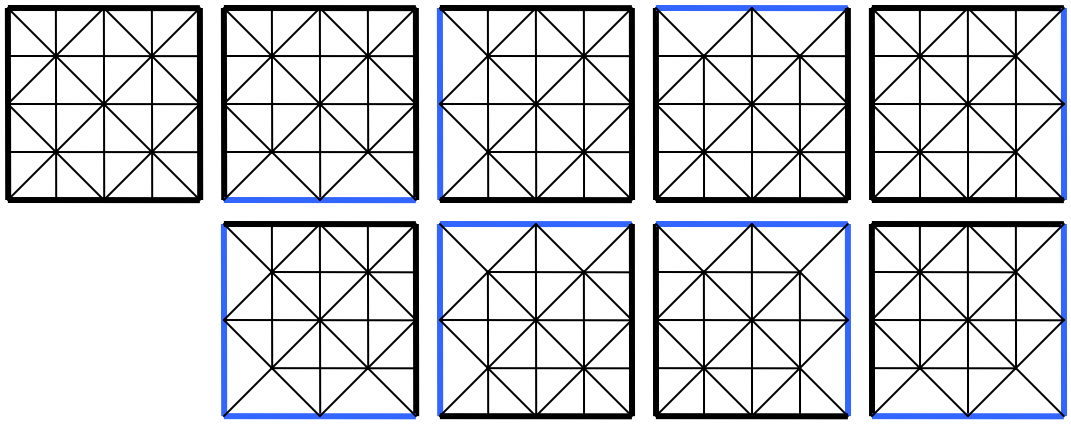


Figure18*. The 9 geometry permutations needed for t-junction free LOD transitions*

5.5 Undergrowth

No matter how advanced shading, texturing and lighting we have on the terrain, it will still not look natural up close due to the inherent limitations of heightfields as geometry (Figure 19).

What we want is detail geometry and meshes to fill up the terrain with undergrowth, grass, plants, stones and debris to create a much richer environment (Figure 20). And since the heightfield geometry and terrain materials and texturing can change due to ground destruction, this detail geometry needs to be able to be updated too.

***Figure19**. Landscape without undergrowth*

***Figure20**. Landscape with undergrowth*

Manually placing individual stones or plants over the whole terrain is not a feasible approach neither from time management nor data management point of view. But in practice, the level designers do not even need or prefer this amount of control.
The undergrowth geometry is also rather small in scale, about 1 m max, and we want it to be very dense with up to a couple of instances per m^2. This makes just storing and loading the instance data (transform) problematic on a large 2 x 2 or 4 x 4 km terrain.

Automatic procedural generation of the placement (procedural instancing) can solve both the content workflow and the memory storage problem.

Procedural generation of instance data can either take place as a pre-process offline or as an on-demand step in the runtime. We choose the latter since it has significantly lower memory and disk storage requirements as well allows us easy regeneration of areas affected by ground destruction dynamically.

5.5.1 Method

In previous games, such as *RalliSport Challenge 2 and Battlefield 2*, we procedurally generated undergrowth instance data based on separate material index CPU textures

top-projected on the terrain. These maps indexed artist-defined undergrowth materials that contained distribution settings such as which meshes to distribute, density (amount per m^2), random scale range, animation settings, etc.

The system worked well but there were three main limitations with the material index maps that we wanted to resolve in Frostbite:

- *Undergrowth materials can not overlap*. Painting an area with a different type of undergrowth is cumbersome and limited since you need to clone the material that was already there, and in the material add the new types of geometry to distribute.
- *Undergrowth materials are fully separate from the underlying terrain materials*. If the terrain textures were repainted to be dirt instead of grass in an area, the undergrowth material index map would have to be repainted manually as well.
- *Resolution and destruction*. With dynamic ground destruction we need to have a much higher resolution of these textures costing memory.

As we now texture, shade and distribute terrain materials and textures through shaders it felt natural to use the same system of procedural shader splatting for the undergrowth generation. Then all the already existing terrain materials could automatically have undergrowth distributed in their specific areas with minimal work on content.

Ground destruction and overlapping materials are also already a part of the general terrain material masking so it is a very good fit.

5.5.2 Generation

Due to the small scale and high density of the undergrowth, we generate and keep only the areas close to the player (and other important viewports) in memory. This is done through a basic grid structure where 16 x 16m undergrowth grid cells are allocated and de-allocated dynamically from a fixed pool of cells when moving around the landscape.

To prevent performance drops when rotating the views quickly with a gamepad or (worse) mouse; the allocation of cells is done on a 2d *xz* distance-basis from the viewport origins instead of when cells are visible in the viewport frustums. A cell viewport frustum check can still be used to separate which cells *need* to be generated as soon as possible, and which *should* be generated to further balance out generation cost over multiple frames.

Each cell contains a list of the undergrowth mesh types in the area and a vertex buffer with the instance data of all instances. The instance data is usually just a 4 x 3 world transformation compressed as fp16 values to save memory and increase GPU performance.

As a cell become visible or is affected by ground destruction, we render out 4-12 material mask values as well as the terrain normal with a top-down projection over the cell area to 2-4 ARGB8888 64x64 simultaneous render targets using MRT (Figure 21).

The shader used is automatically generated offline in a similar manner to the terrain shader compositing shaders.

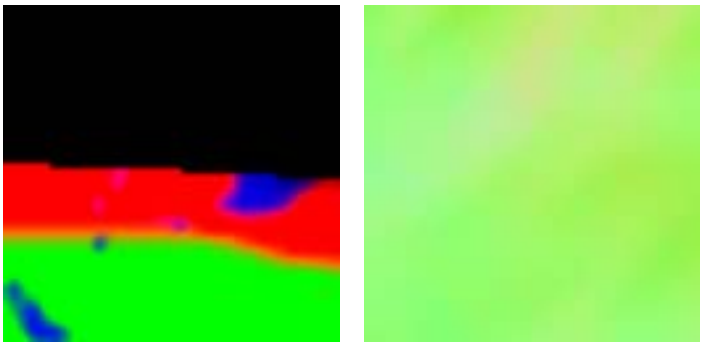


***Figure21**. 4-channel undergrowth mask texture where black is no undergrowth (left). Undergrowth cell normal map from terrain (right)*

When the textures have been rendered by the GPU, we lock them (or in Direct3D10 copy them to a staging texture) for processing by a CPU thread or an SPU.

The CPU processing scans through the texture for every undergrowth material in a randomly jittered grid pattern over the cell space where the grid size is dependent on the material density setting. At every sample point the material mask texture is read and Russian roulette is played to determine if an undergrowth instance should be placed at that point.

If it passes, the terrain normal map is then used to either rotate or skew the instance to fit the ground.

The randomly jittered grid pattern works by generating uniform points on a grid and randomly offsetting the points with a maximum of a half cell length, giving a uniform but varied distribution. This reduces overlap of instances compared to ordinary pseudo-random distribution which is important both visually and for performance for materials such as grass.

To get deterministic results when generating pseudo-random numbers within a cell, the cell position in the grid structure is hashed and used as a seed. This is important both on the local client when regenerating cells but also when running multiple clients of the network so that everybody sees the same geometry.

In Direct3D10, the whole generation step can be moved to the GPU using Stream Output ([Blythe06]) to offload the CPU and to reduce latency in the generation.

5.5.3 Rendering

After the undergrowth cells have been generated, rendering them is easy.

The undergrowth meshes are low-poly meshes with arbitrary surface shaders (Figure 22) that are rendered using stream instancing. They use alpha-testing or alpha-to-coverage and are rendered in front-to-back order on a per-cell basis to improve hierarchical Z-cull, though the amount of small detail in the textures makes hierarchical Z-cull not very effective.

Through the use of the surface shader framework and runtime the undergrowth will receive the same per-pixel lighting and shadowing as any other surfaces in the engine which looks good makes it easier for it to blend in with the rest of the environment.

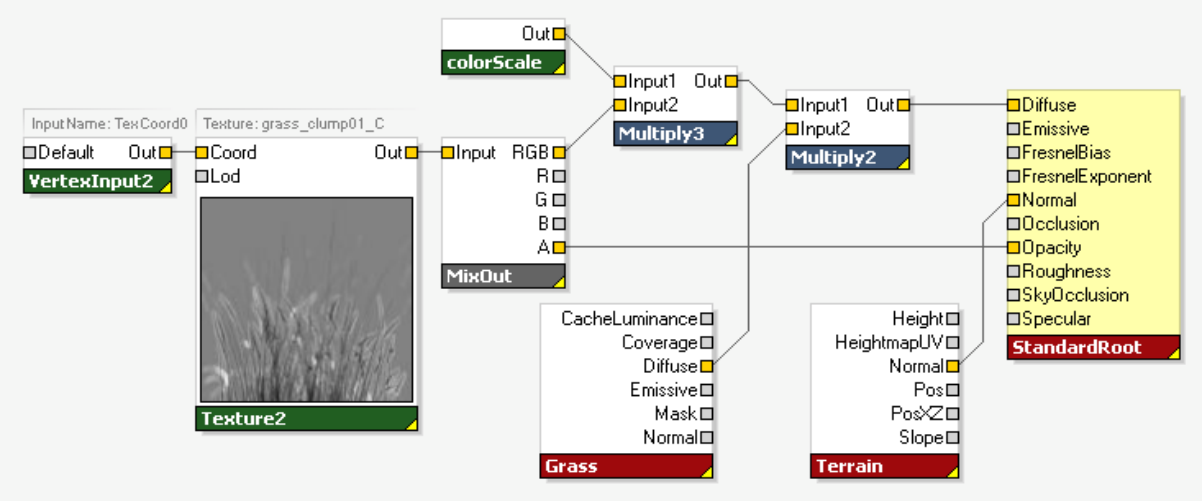


***Figure22**. Undergrowth surface shader for a grass mesh. Blends in with the terrain by compositing its color map with the diffuse color from the actual terrain grass shader. Lighting uses the normal of the heightfield to look the same as the terrain.*

5.6 Conclusions

We have presented a flexible framework and technique for terrain rendering called *Procedural Shader Splatting* where graph-based surface shaders control terrain texture compositing and distribution to allow terrain materials to be individually specialized to balance performance, memory, visual quality and workflow.

The technique allows us to support dynamic heightfield modification for ground destruction while keeping visual quality high both in a distance and close up and memory usage low.

Procedural instancing of undergrowth is integrated into the system and using the terrain material distribution and shaders is a very powerful tool and easy way to add visual detail for a low cost in both memory and content creation.

There are however a few inherent drawbacks with the technique:

- *Performance*. Since almost all of the texture compositing is done in the shaders in runtime instead of stored in offline created color maps, this approach is in general more costly(due to shader instruction count and number of texture fetches) than for a traditional fixed scheme such as in *Battlefield 2*.
- *Complex workflow*. While the artists still can choose to paint mask textures and color maps, to really utilize the system they need to combine that with procedural shading which is unfamiliar and not fixed cost as textures. On the other hand,

procedural elements can be more easily shared and reused across multiple terrains.

The flexibility built into the technique and framework makes it a great scaleable platform to integrate interesting shading techniques and texturing schemes in the future.

5.7 References

[AT07] ANDERSSON, J., TATARCHUK, N. 2007. Frostbite Rendering Architecture and Real-time Procedural Shading and Texturing Techniques. AMD Sponsored Session. GDC 2007. March 5-9, 2007, San Francisco, CA. http://ati.amd.com/developer/gdc/2007/Andersson-Tatarchuk-FrostbiteRenderingArchitecture(GDC07_AMD_Session).pdf

[BLOOM00] BLOOM, C. 2000. Terrain Texture Compositing by Blending in the Frame-Buffer (a.k.a. "Splatting" Textures). Nov 2, 2000. http://www.cbloom.com/3d/techdocs/splatting.txt

[BLYTHE06] BLYTHE, D. 2006. The Direct3D 10 system. In proceedings of ACM Transactions on Graphics (SIGGRAPH'06 Conference Proceedings), pp. 724-234, Boston, Massachusetts.

[DALLAIRE06] DALLAIRE, C. 2006. Binary Triangle Trees for Terrain Tile Index Buffer Generation. Gamasutra article. http://www.gamasutra.com/features/20061221/dallaire_01.shtml

[TATARCHUK06] TATARCHUK, N. 2006. Dynamic parallax occlusion mapping with approximate soft shadows. In proceedings of AMD SIGGRAPH Symposium on Interactive 3D Graphics and Games, pp. 63-69, Redwood City, CA.

[TATARCHUK07] TATARCHUK, N. 2007. The Importance of Being Noisy: Fast, High Quality Noise. Conference Session. GDC 2007. March 5-9, 2007, San Francisco, CA. http://ati.amd.com/developer/gdc/2007/Tatarchuk-Noise(GDC07-D3D_Day).pdf

[TERRAGEN*] TERRAGEN by Planetside Software. http://www.planetside.co.uk/terragen/

[WEI04] WEI, L. 2004. Tile-Based Texture Mapping on Graphics Hardware. In proceedings of ACM *SIGGRAPH/Eurographics conference on Graphics Hardware*, pp. 55-63. Grenoble, France.

Chapter 6

Dynamic Deformation Textures

GPU-Accelerated Simulation of Deformable Models in Contact

Nico Galoppo[8] Miguel A. Otaduy[9] Paul Mecklenburg[10]
UNC Chapel Hill ETH Zurich UNC Chapel Hill

Markus Gross[11] Ming C. Lin[12]
ETH Zurich UNC Chapel Hill

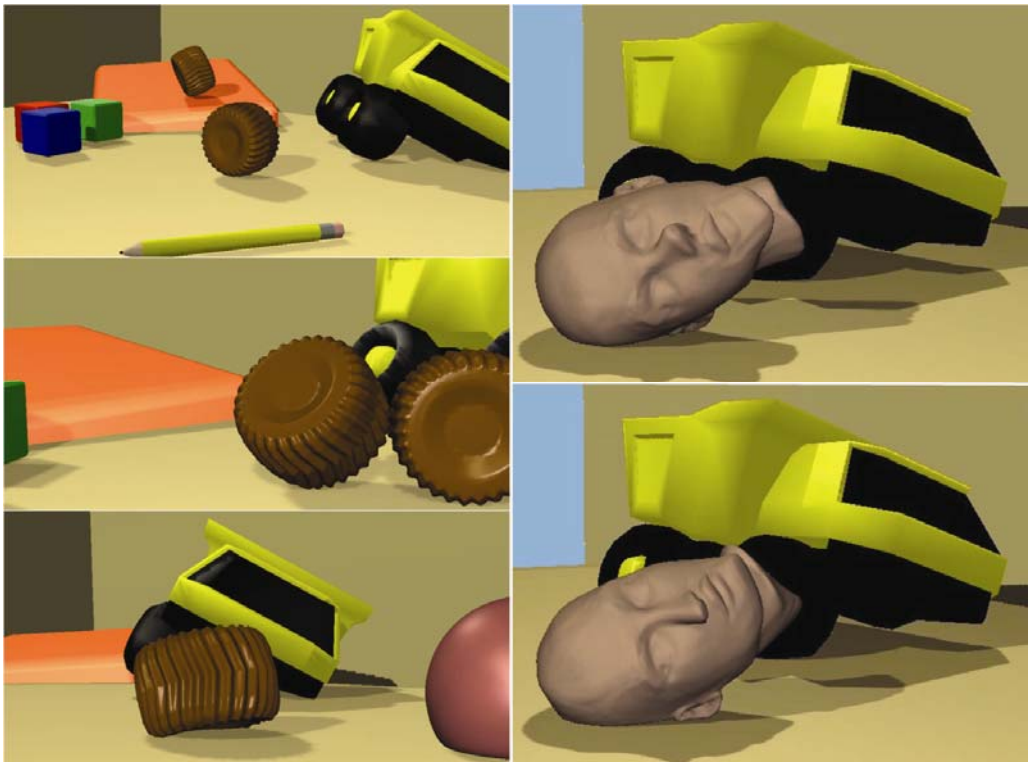


Figure 1. Soft Object Interaction in a Dynamic Scene. *Deformable objects roll and collide in the playground. In these simulations, over 15,000 contacts were processed per second.*

[8] email: nico@unc.edu
[9] email: otaduy@inf.ethz.ch
[10] email: prm@cs.unc.edu
[11] email: grossm@inf.ethz.ch
[12] email: lin@cs.unc.edu

6.1 Introduction

We present an efficient algorithm for simulating contacts between deformable bodies with high-resolution surface geometry using *dynamic deformation textures* (D2Ts), which reformulate the 3D elastoplastic deformation and collision handling on a 2D parametric atlas to reduce the extremely high number of degrees of freedom arising from large contact regions and high-resolution geometry. Such computationally challenging dynamic contact scenarios arise when objects with rich surface geometry are rubbed against each other while they bounce, roll or slide through the scene, as shown in Figure 1.

We simulate real-world deformable solids that can be modeled as a rigid core covered by a layer of deformable material [TW88], assuming that the deformation field of the surface can be expressed as a function in the parametric domain of the rigid core. Examples include animated characters, furniture, toys, tires, etc.

Our mathematical formulation of dynamic simulation and contact processing, along with the use of dynamic deformation textures, is especially well suited for realization on commodity SIMD or parallel architectures, such as graphics processing units (GPU), Cell processors, and physics processing units (PPU). More in particular, the following key concepts contribute to the mapping of our algorithm to the GPU architecture, resulting in the effectiveness and efficiency of our algorithm:

- We reformulate the 3-dimensional elastoplastic deformations and collision processing on 2-dimensional dynamic deformation textures. This mapping is illustrated in Figures 3 and 8, with the 2D computational domains indicated by T and D. There is a natural mapping between the computational domains and graphics hardware textures.

- Using a two-stage collision detection algorithm for parameterized layered deformable models, our proximity queries are scalable and output-sensitive, i.e. the performance of the queries does not directly depend on the complexity of the surface meshes. We perform high resolution collision detection with an image based collision detection algorithm, implemented on the GPU.

- By decoupling the parallel update of surface displacements and parallel constraint-based collision response from the update of the core DoFs, we provide fast and responsive simulations under large time steps on heterogeneous materials. We have implemented this parallel implicit contact resolution method on the GPU, thereby exploiting the inherent parallelism of the GPU architecture.

A common thread throughout the design of our algorithm is the desire to minimize data transfer between GPU and CPU while maximizing the parallel computation power of the GPU. In our algorithm, surface simulation, collision detection and rendering are all performed by the GPU exploiting our common D2T model representation. The surface deformation is simulated very efficiently on the GPU in fragment programs and updated in texture memory. A dynamically changing position texture is thus available for collision detection and rendering.

6.2 Algorithm Overview and Parallel Implementation

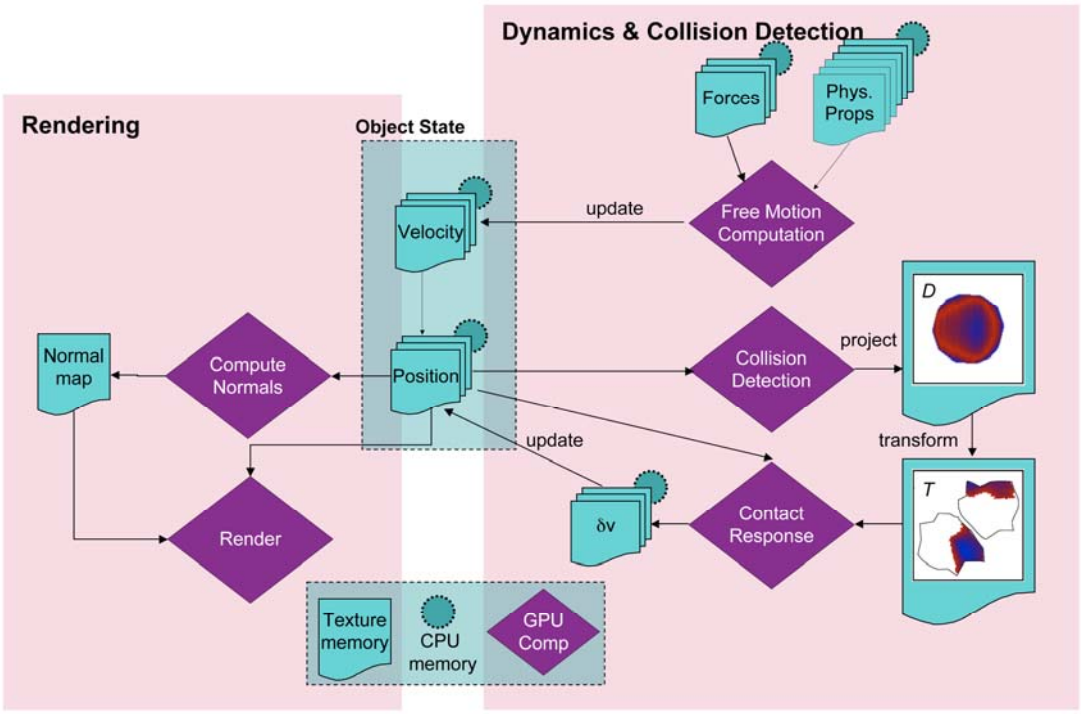


Figure 2. *Algorithm Overview.*

The implicit formulation of the dynamic motion equations and collision response yields linear systems of equations with dense coupling between the core and elastic velocities. However, we can formulate the velocity update and collision response in a highly parallelizable manner [GOM+06]. In Figure 2 we illustrate how our algorithm for simulating and rendering deformable objects in contact using dynamic deformation textures maps to the GPU. Algorithm 2.1 shows a more detailed breakdown of the steps of the *dynamics* computations. We refer the interested reader to [GOM+06] for details of the equations. Let s in Algorithm 2.1 denote the operations that are performed on small-sized systems (i.e., computations of core variables, and low resolution collision detection). The remaining operations are all executed in a parallel manner on a large number of simulation nodes. Specifically, T refers to operations to be executed on all simulation nodes in the dynamic deformation texture T, D refers to operations to be executed on texels of the contact plane D, and T_D refers to operations to be executed on the colliding nodes. As highlighted in Algorithm 2.1, all operations to be executed on simulation nodes (indicated by T, T_D and D) can be implemented with parallelizable computation stencils on the GPU, as indicated in Figure 2 by purple diamond boxes. Moreover, due to the regular meshing of the deformable layer produced by dynamic deformation textures, the computation stencils are uniform across all nodes; hence they are amenable to implementation on a streaming processor such as the GPU. In Section 6.5 we will illustrate this concept for representing and computing sparse matrix multiplications in step 2 of Algorithm 2.1.

ALGORITHM 2.1 Summary of Our *Dynamics* Simulation Algorithm

	COLLISION-FREE UPDATE	
1.	Evaluate forces	T
2.	Solve the sparse linear systems $\tilde{\mathbf{M}}_e\mathbf{y} = \tilde{\mathbf{F}}_e$ and $\tilde{\mathbf{M}}_e\mathbf{Y} = \tilde{\mathbf{M}}_{ec}$ ([GOM+06]), using a Conjugate Gradient solver [GV96]	T
3.	Update core velocities $\mathbf{v}_c^-$	s
4.	Update elastic velocities $\mathbf{v}_e^-$	T
5.	Perform a position update $\mathbf{q}^- = \mathbf{q}(t) + \Delta t\mathbf{P}^+\mathbf{v}^-$	T
	COLLISION DETECTION	
6.	Execute low-resolution collision detection	s
7.	Compute penetration depth and contact normal	D
8.	Map contact information to the dynamic deformation textures	T
	COLLISION RESPONSE	
9.	Invert the block-diagonalized full-rank matrix $\mathbf{J}_e\tilde{\mathbf{M}}_e^{-1}\mathbf{J}_e^T$	T_D
10.	Solve for Lagrange-multipliers λ using the Sherman-Morrison-Woodbury formula	T_D
11.	Repeat steps 3 and 4 to obtain the collision impulse $\delta\mathbf{v}$	
12.	Compute friction impulse	T_D
13.	Perform a position update $\mathbf{q}(t+\Delta t) = \mathbf{q}^- + \Delta t\mathbf{P}^+(\delta\mathbf{v})$	T
	CONSTRAINT CORRECTION	
14.	Repeat collision detection steps 6 to 8	
15.	Apply constraint correction	T_D

Figure 2.1. *Algorithm Overview.*

For the execution of collision detection, we exploit image-based computations on the GPU. As the dynamically deforming surface is updated in texture memory directly, its state is available to the collision detection module without requiring an expensive update from the CPU host. The computations of per-texel penetration depth and contact normal are performed by orthonormal projection of the geometry, as described in Section 6.6.2.

Finally, after computing collision response of steps 6-15 and updating the position D2T in texture memory, the state of the surface is readily available for rendering. Section 6.7 describes how the deforming mesh is drawn to the screen using our D2T model representation.

6.3 Dynamic Deformation Textures

We encode the state of the deformable surface in *dynamic deformation textures* or D2Ts. A D2T consists of a texture atlas, with potentially multiple patches (Figure 3), in which each texel *(s, t)* that falls within the patches implicitly represents a vertex on the

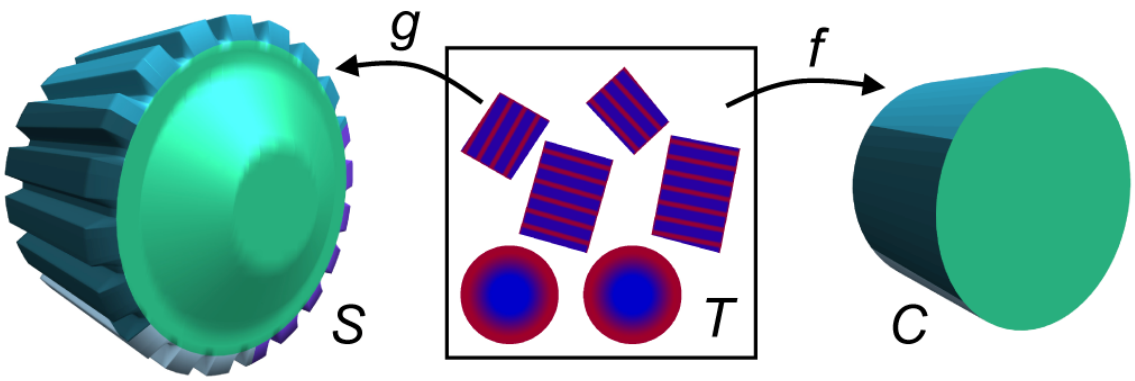


Figure 3. Deformable Object Representation. *Deformable surface S (*52K *triangles) and core C of a gear, showing the patch partitioning of its texture atlas. The dynamic deformation texture T (*256×256*) stores the body space positions on the surface. The gear contains* 28K *simulation nodes on the surface and* 161K *tetrahedra, allowing the simulation of highly detailed deformations.*

surface. These texels are also referred to as valid texels. Each texel $(s,t) \in T$ maps to two corresponding points $f(s,t)$ and $g(s,t)$ on the surfaces of the core and the deformable object as indicated in Figure 3. The regular sampling of T and the correspondence of surface points define implicitly a meshing of one layer of tetrahedral elements, as shown in Figure 4. By applying classical approximation methods such as FEM, the deformation field in the deformable layer can be approximated from the values at a finite set of nodes. Since there is never any deformation at points on the core, the deformation field can be approximated entirely from the values at surface nodes. Effectively, each texel $(s,t) \in T$ maps to a simulation node $g(s,t)$ in the FEM discretization. Simulation variables defined per-node, such as velocities, forces, mass and stiffness values, etc. can also be stored in the D2T texture atlases. Note that the implicitly defined texture-based meshing is not consistent at patch boundaries, which requires special yet simple treatment as discussed in Section 6.5.3.

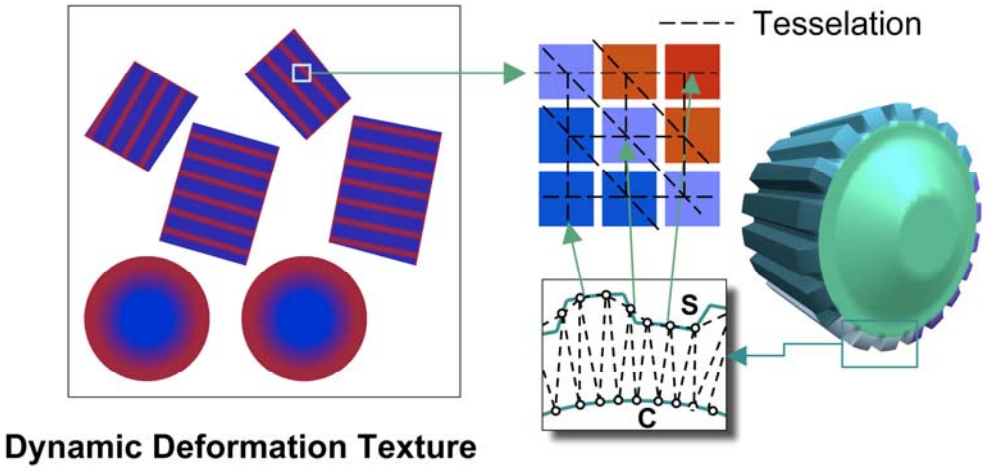


Figure 4*. Dynamic deformation texture representation and implicit tessellation.*

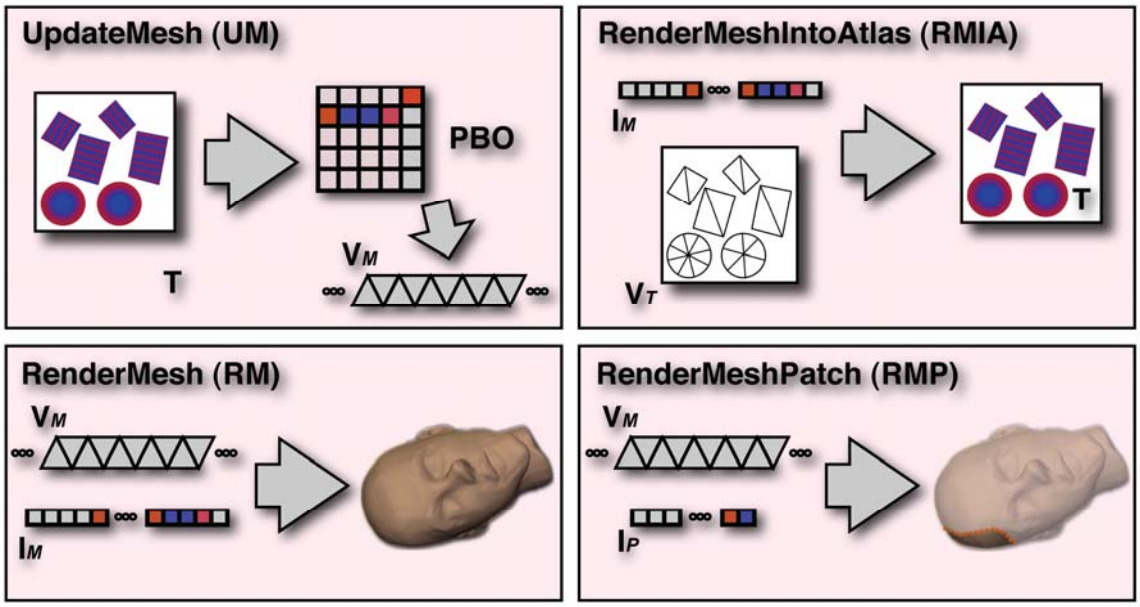


***Figure 5**. Rendering Blocks*

In a preprocessing step, we tessellate the mesh from the vertex connectivity that is implicitly defined by the texel grid in the D2T texture (see Figure 4). The implicitly defined triangle strips are encoded in a vertex index list I_M. Additional triangle strips are constructed to patch or *zipper* [TL94] the mesh at the cuts along the patch boundaries.

6.4 Basic Rendering Blocks

In this section, we define a few basic blocks that are used to render the deforming mesh to the screen and into the collision and simulation domains. Note that our representation of a deformable mesh is carefully chosen such that we avoid expensive GPU readback or host upload at all times. Therefore, the mesh topology is stored in a static index buffer on the GPU and all surface vertex position data is stored in texture memory, while the surface deformation simulation is computed using fragment programs on the GPU. The blocks are illustrated schematically in Figure 5.

UpdateMesh (UM) This block is used to update a dynamic vertex buffer V_M with the deformed surface vertex positions after each time step in the simulation. One approach to render the deforming surface, given the dynamic deformation texture T on Shader Model 3.0 hardware, is to fetch the positions from T in the vertex shader. Each vertex can then be displaced according to the current position stored in T. Unfortunately, the less powerful vertex processing pipe and slow vertex-stage texture fetches of non-unified GPU architectures can make this approach a bottleneck, especially because the UpdateMesh block will be used multiple times for the same snapshot in time. It would be wasteful to repeat the displacement in the vertex shader for collision detection, shadow map generation and multiple final render passes.

Therefore, we use the OpenGL PBO extension to copy the D2T texture T to a pixel buffer object that can later be interpreted by the OpenGL API as a vertex buffer object

VM (see code snippet 1 in Listing 1). This technique is often referred to as the PBO/VBO approach to render-to-vertex-array. This data copy is efficient because it is between two GPU memory areas; there is no data copy to or from the host. Note that in this approach not all memory locations in the PBO contain valid vertex data, because not all texels in T are valid (Section 6.3). The vertex indices in I_M are assigned such that they index into the correct location of the PBO. We store the triangle list in the static index buffer I_M; thus we can render the vertices without any vertex bandwidth overhead with an indexed draw call (`glDrawElements()` for the OpenGL API).

RenderMesh (RM) This block is the encapsulation of the vertex processing stage on the GPU, when rendering a deformable mesh. Given the index buffer I_M and the dynamic vertex buffer V_M, the deforming geometry can be rendered efficiently with a single indexed draw call.

RenderMeshPatch (RMP) This block is identical to the RenderMesh block, except that the input index list I_P is not static. In this case, we render only a subset of the mesh's triangles by sending the vertex index list at each frame. As it is only a limited number of triangles, this is not a significant overhead.

RenderMeshIntoAtlas (RMIA) and RenderPatchIntoAtlas (RPIA) In many simulation parts of our algorithm, it is required to render values defined on the surface of the mesh into the D2T texture atlas. This can easily be achieved by the RenderMeshIntoAtlas block. We store the D2T texture coordinates as positions a separate (static) the vertex buffer V_T. Therefore, through the use of the identity matrix as the model-view-projection matrix, we achieve the desired rasterization into the D2T texture atlas. The same operation can also be performed for a subset of the mesh triangles. We call this block RenderPatchIntoAtlas.

6.5 Simulation of Surface Deformations

As mentioned in Section 6.3, we perform dynamic simulation of the surface deformable object in the domain of the *dynamic deformation texture* (D2T). The goal of the dynamic simulation part of the algorithm is to compute the global motion of objects (i.e. the rigid motion of the core C) and to compute how the surface S deforms under influence of forces and accelerations. In practice, we can do this very efficiently exploiting the parallelism on the GPU in fragment programs while rendering the results directly to the dynamically changing D2T position texture which can then be used for collision detection and rendering. The only information communicated between CPU and GPU are a few small state changes, typically 6-tuples or 3×3 matrices. These state changes are required for updates that are related to the rigid transformation modes of C and for transferring forces and accelerations that are due to dynamic coupling between the deformable surface and the core.

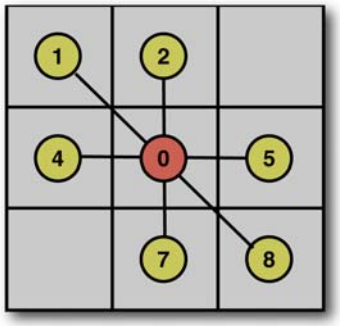


Figure 6. *A texel in the D2T defines a simulation node. The figure shows its neighborhood in the FEM mesh. Its 6 neighbors and itself give rise to 7 non-zero blocks per block row in the stiffness matrix, as shown in Figure 7.*

This section will only touch on a few concepts and simple shaders that are being used to map step 2 in Algorithm 2.1 to the deformation simulation to the GPU pipeline. In reality, our implementation of all dynamics steps in Algorithm 2.1 consists of 50-100 different shaders that compute the different steps in the dynamics equations and contact handling. For details on the dynamics equations and on the theoretical fundamentals of our simulation algorithm, please refer to [GOM+06].

6.5.1 Velocity and Position Updates

At the core of the dynamics simulation of a mesh with n vertices, a large linear system $\mathbf{Ax} = \mathbf{b}$ has to be solved at each time step to compute the velocity at the next time step, where $\mathbf{x}$ and $\mathbf{b}$ are vectors of size n. The matrix $\mathbf{A}$ is a symmetric, positive definite sparse block matrix, where the non-zero blocks are 3×3 matrices (Figure 7). Such a system can be solved with any variant of the conjugate gradients (CG) solver [She94]. The conjugate gradients method is an iterative solver and a very important building block of CG are sparse matrix multiplications of the form $\mathbf{y} = \mathbf{Ax}$.

In the remainder of this section, we will explain how $\mathbf{A}$ is stored and how these sparse matrix multiplies are performed in a fragment program.

6.5.2 Sparse Matrix Representation and Multiplication

The vectors $\mathbf{x}$ and $\mathbf{y} \in R^{3n}$ both define vector values (3-tuples) at each vertex. We already know from Section 6.3 that we can store those values at valid texels in the D2T texture atlas. We can also map $\mathbf{A}$ to the D2T atlas as follows. Each block row of $\mathbf{A}$ defines seven 3×3 blocks, one for each neighbor of a given vertex (or texel in the D2T) as shown in Figure 6. Hence, we can store $\mathbf{A}$ in 21 RGB textures where each texture stores a 3×1 row of a 3×3 block (Figure 7). Due to the limited number of texture samplers that can be bound to a fragment program within a pass, the actual sparse matrix multiplication has to be performed in two passes. Mathematically, this corresponds to the following transformation: $\mathbf{Ax} = \begin{bmatrix} \mathbf{A}_l & \mathbf{A}_r \end{bmatrix}\mathbf{x} = \mathbf{A}_l\mathbf{x} + \mathbf{A}_r\mathbf{x}$. In the second

pass, the result of $\mathbf{A}_i\mathbf{x}$ is passed in from the first pass. Code Snippet 2 in Listing 2 shows the setup and invocation of the passes, while Fragment Programs 1 and 2 (Listing 3 and 4) show the implementation in the fragment processor. Note that if $\mathbf{x}$ is an $n\times 3$ matrix instead of a vector of size n, the result is an $n\times 3$ matrix. This can still be achieved in 2 passes by rendering to multiple render targets simultaneously, storing 3 columns instead of 1.

This approach of matrix multiplies is very efficient on parallel streaming processors such as current GPUs, because there is no branching or pointer chasing involved. Moreover, our mapping of the sparse matrix to the D2T atlas exploits the GPU texture caching architecture in two ways. First, due to tile based rendering, neighboring values fetched from $\mathbf{x}$ and $\mathbf{A}$ in one fragment are conveniently pulled into cache for neighboring fragments in the same tile. Second, fetching a value pulls in other neighboring values from $\mathbf{x}$ and $\mathbf{A}$ that are required in the same fragment program for free.

6.5.3 Patch Boundary Handling

In the previous section, we have neglected the fact that, at patch boundaries in the D2T, not all neighboring texels are valid texels. One solution could be to flag boundary texels in some way and use branching that is available in current GPU hardware, but this is not very efficient because the boundaries are not coherent fragment blocks. Better approaches are to rasterize and handle the boundary texels separately with a separate fragment program [Har05] or to guarantee that all neighbors are valid. We have taken the latter approach. We adapt a method by Stam [Sta03] for providing accessible data in an 8-neighborhood to all nodes located on patch boundaries. Before every sparse matrix multiplication step in the algorithm, we fill a $\sqrt{2}$ -texel-width region on patch boundaries by sampling values on the adjacent patches. In practice, for each deformable model and D2T atlas, we maintain a list of thin quads that have texture coordinates assigned that map to locations of neighboring surface points across boundaries in the D2T texture atlas.

6.6 Texture-Based Collision Detection

In this section we describe our texture-based collision detection algorithm for deformable surfaces with high-resolution geometry. For this course, we focus on the image-space detection of surface interpenetration and how these are mapped to the texture-based simulation domain. We opted for a GPU-accelerated image-space algorithm because it exploits the surface position data that is stored and simulated in fast texture memory. Therefore, we avoid the need to transfer large amounts of mesh position data between CPU and GPU, which could easily become a bottleneck for our system otherwise. For the formulation of the dynamic system to solve contact response, we refer the reader to [GOM+06].

We propose to perform collision detection between two deformable objects A and B in two steps:

1. Identify contact regions with object-space techniques using low-resolution proxies of the objects.

2. Compute contact information in the contact regions using image-space techniques and high-resolution displacement fields.

A similar approach has been exploited for estimating the penetration depth value between rigid, high-resolution objects [OJSL04], but we perform collision handling of deformable objects and compute contact information for many colliding surface points.

6.6.1 Object-Space Collision Detection

For the first step, we assign to each object a low-resolution proxy, i.e. a low-polygon count approximate convex hull of the hi-res geometry. We identify potentially colliding objects using a fast collision detection algorithm for convex objects [EL00] that identifies low-resolution proxies that are within a user specified tolerance distance of each other. In this step, we identify patches of the proxies that are closer together than a distance tolerance. Given a contact region between core surface patches $C_A \in R^3$ and $C_B \in R^3$, we identify a contact plane $D \in R^2$ as the plane passing between the contact points and oriented according to the contact normal.

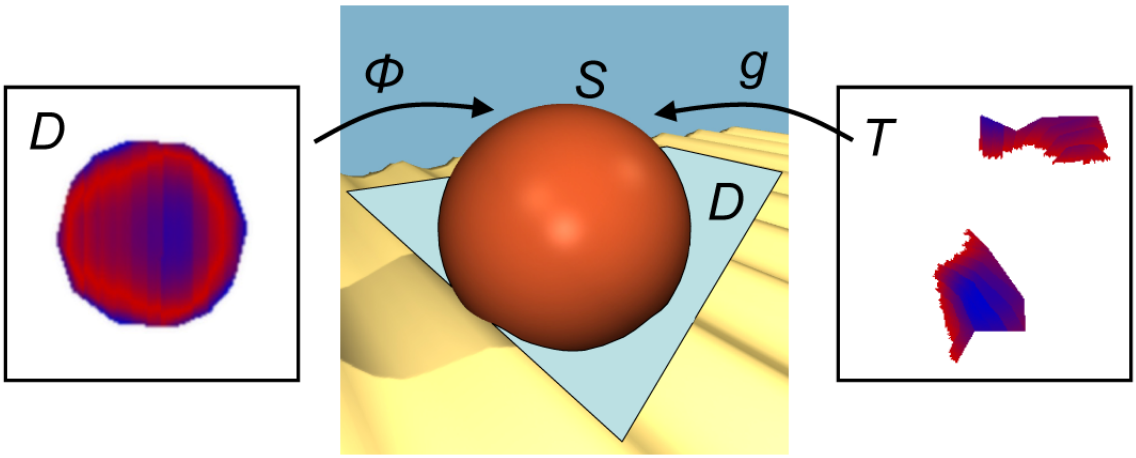


Figure 8. Texture-Based Collision Detection Process. *Center: A sphere S collides with a textured terrain. Left: Contact plane D for texture-based collision detection, and mapping* $\phi : D \to S$. *The contact plane shows the penetration depth. Right: Dynamic deformation texture T, and mapping* $g : T \to S$. *The penetration depth is projected from D to T, and is available for collision response.*

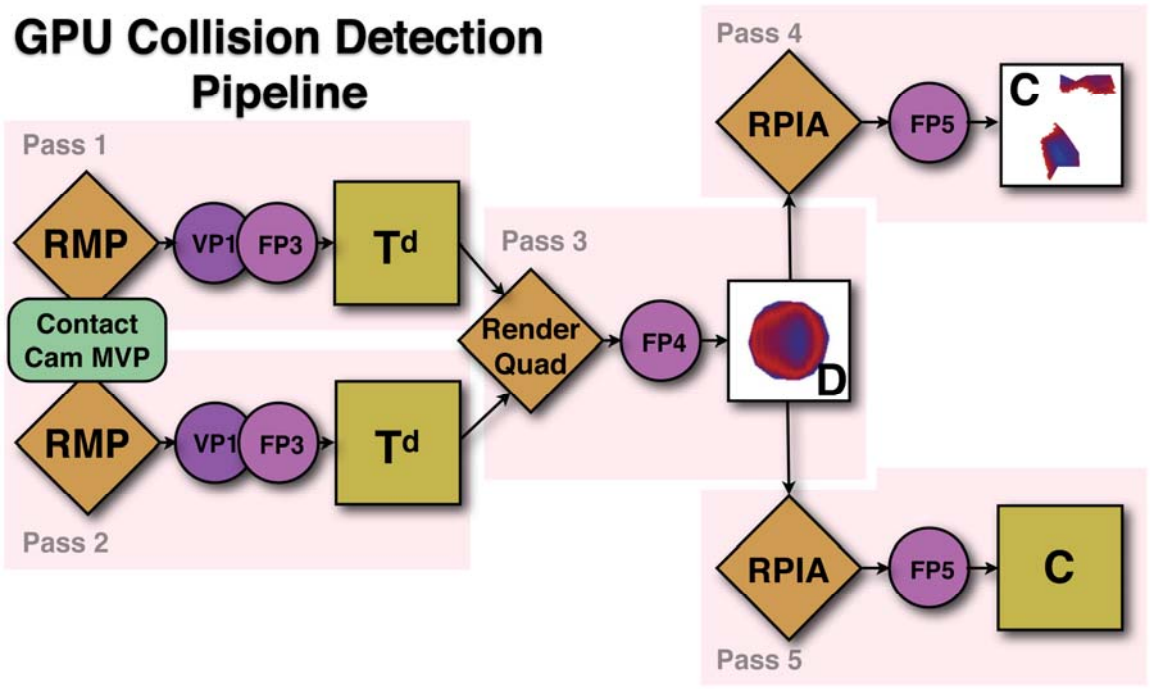


Figure 9. *Schematic overview of the pipeline of our GPU-based collision detection algorithm, composed out of 5 passes.*

6.6.2 Image-Space Collision Detection

The second step in our algorithm is accelerated by the GPU. This stage utilizes the RenderMeshPatch block (Section 6.4). We restrict the draw call to the triangles that form the potentially colliding surface patch. Our image-based algorithm consists of three substeps that are implemented by five rendering passes per pair of potentially colliding surface patches (Figure 9).

In the first two passes, we perform a projection step for each potentially colliding surface patch. We set up an orthographic projection which we call the contact camera. The contact camera is carefully positioned such that it looks along the normal of the contact plane D and such that the projections C_A and C_B capture the full extent of the contact area of a pair of potentially colliding surface patches S_A and S_B (Figure 10). Vertex Program 1 (Listing 5) and Fragment Program 3 (Listing 6) are used to rasterize the distance from the eye directly into textures T_A^d and T_B^d. Note that we enable front-facing triangles while rasterizing S_A into T_A^d and back-facing triangles while rasterizing S_B into T_B^d. In the third pass, we capture the areas of interpenetrating surface patches by constructing texture D from projections T_A^d and T_B^d. For each texel $(u,v) \in D$, we perform high-resolution collision detection by testing the distance between points $C_A(u,v) \in S_A$ and $C_B(u,v) \in S_B$ along the normal direction of D. If the points are penetrating, we identify a contact constraint and we compute the contact normal n as the average surface normal. We also approximate the penetration depth as $d = \mathbf{n}^T(T_B(u,v) - T_A(u,v))$ for applying constraint correction. This approximation is

affected by texture distortion as the surface deforms, but we have not found noticeable errors in our examples. In practice, as shown in the middle of Figure 9, we render a full-screen quad of the size of T_A^d and T_B^d into D, while Fragment Program 4 (Listing 7) computes the difference in distances. Positive values indicate penetration in the projection as indicated by the red regions on the left in Figure 8.

Note that we also write the triangle ID of the current fragment to D. These IDs are used in the next pass to check whether a rasterized texel of the D2T is originating from the triangle whose fragments were rasterized into D and not from a triangle that maps to the same texel in D (see Figure 10).

Recall that the deformation of the sphere is stored in the two-dimensional texture atlas T called dynamic deformation texture (D2T). This texture atlas is shown on the right in Figure 8. We compute dynamic contact response in this domain. Therefore, the collision information in texture D has to be transferred to the dynamic deformation texture T via a mapping that is the combination of the inverse of the orthogonal contact projection with the D2T texture atlas mapping. In practice, this step is performed by the two last passes of our algorithm. These passes render each potentially surface geometry again using the RenderMeshIntoAtlas block (Section 6.4) into the D2T domain. We set up the texture matrix to perform the correct mapping while fetching values from texture D. The required texture matrix set up is completely analogous to the typical setup for traditional shadow mapping. Here, the contact camera model-view-projection matrix takes the place of the light's model-view-projection matrix. Code Snippet 3 in Listing 9 shows the code that is used for this setup. Fragment Program 5 (Listing 8) shows the pixel shader code of the last two passes, one shader per object.

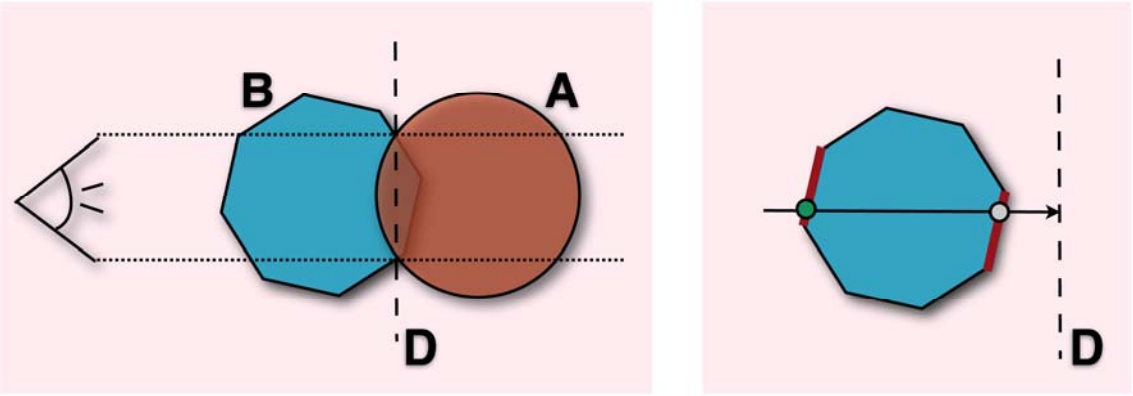


***Figure 10. Left**: The contact camera is set up with an orthogonal projection perpendicular to the contact plane D. **Right**: Multiple surface points may map to the same location on D. When texels in the D2T are tagged as colliding, a check is required which triangle (of the two red triangles) the rasterized fragment belongs to, in order to avoid tagging the green surface point as colliding.*

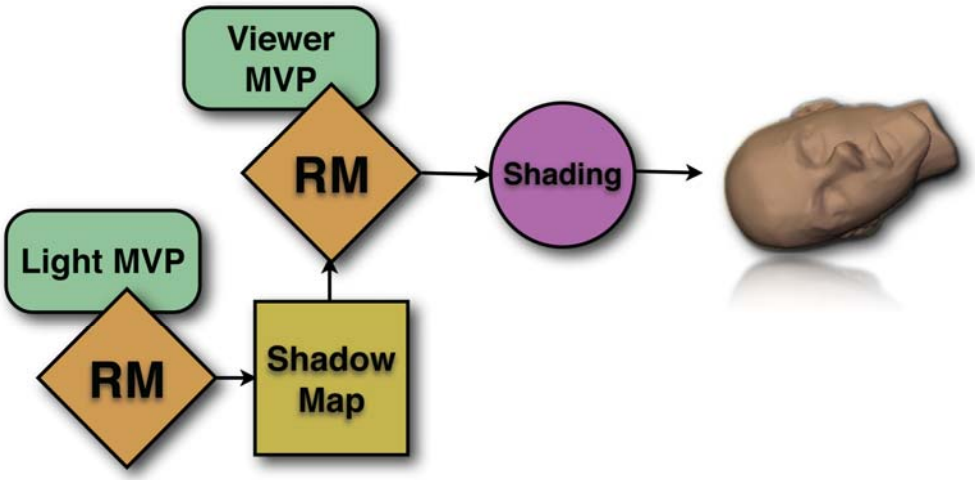


Figure 11. *Rendering pipeline. Note that the RenderMesh (RM) block utilizes the vertex stream with normals generated as in Figure 12.*

6.7 Rendering

Using the RenderMesh block defined in Section 6.4, rendering a deformable mesh represented by D2T position textures and the additional data structures described in Section 6.3 and Section 6.4 is relatively straight forward (see Figure 11). A standard fragment program that computes per-pixel shading is plugged into the pipeline and the RenderMesh block can also be used to generate a standard shadow map.

The only missing piece of information are the vertex normals. As the geometry is deforming, normals have to be recomputed at each frame (or each few frames). There are two approaches possible. On Shader Model 4.0 (DirectX10) compatible hardware, the normals can be computed in a geometry shader provided that an appropriate triangle adjacency list is sent to the GPU. Alternatively, on older hardware, one can generate a normal map using the D2T texture atlas. This process is illustrated in Figure 12 along with Fragment Program 6 (Listing 10). Here, as for sparse matrix multiplication in Section 6.5.2, the input D2T texture has to be augmented with replicated position information along the patch boundaries. This ensures that each D2T texel neighborhood is valid and can be sampled to approximate the corresponding vertex normal. The normals vertex buffer can be updated with the normal map using the PBO technique that was also used when updating the position vertex buffer in Section 6.4.

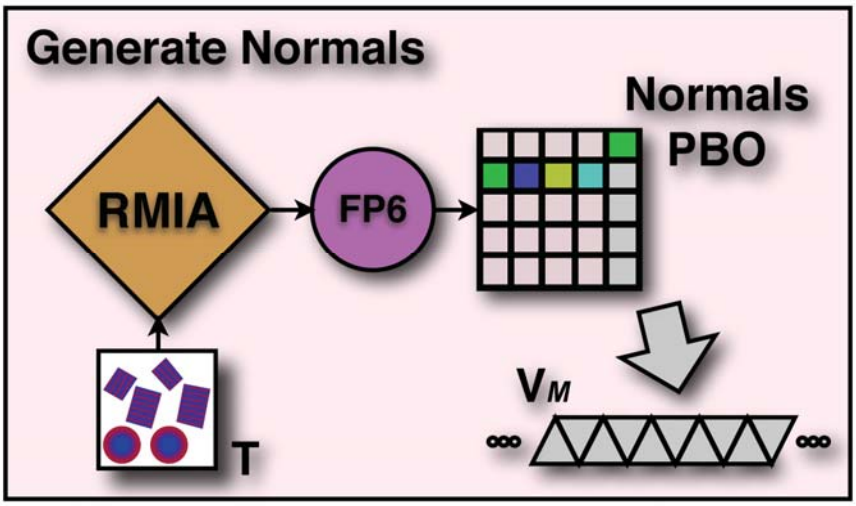


Figure 12. *Normal Generation block. A normals PBO is generated and then copied to the normal vertex buffer.*

6.8 Results

In Figure 1, we show a scene where deformable tires with high-resolution features on their surfaces roll, bounce, and collide with each other. This simulation consists of 324K tetrahedra and 62K surface simulation nodes. Such high resolution enables the simulation of rich deformations, as shown in the accompanying video. All contacts on the surface have global effect on the entire deformable layer, they are processed simultaneously and robustly. Without any precomputation of dynamics or significant storage requirement, we were able to simulate this scene, processing over 15,000 contacts per second, on a 3.4 GHz P4 with NVIDIA GeForce 7800.

Our approach is considerably faster than other methods that enable large time steps, such as those that focus on the surface deformation and corotational methods that compute deformations within the entire volume, with more stable collision response. Our approach can also handle many more contact points than novel quasi-rigid dynamics algorithms using LCP [PPG04], while producing richer deformations, between moving objects (Figure 13).

Acknowledgements

This work is supported in part by ARO, NSF, ONR, DARPA/RDECOM and the Swiss NSF. We'd like to thank the GAMMA group in Chapel Hill and the CGL in Zurich for their support; we'd also like to thank Natalya Tatarchuk (AMD Graphics) for organizing the course, Kenny Erleben (DIKU) for help with tetrahedral meshing and Mark Harris (NVIDIA) for GPGPU support.

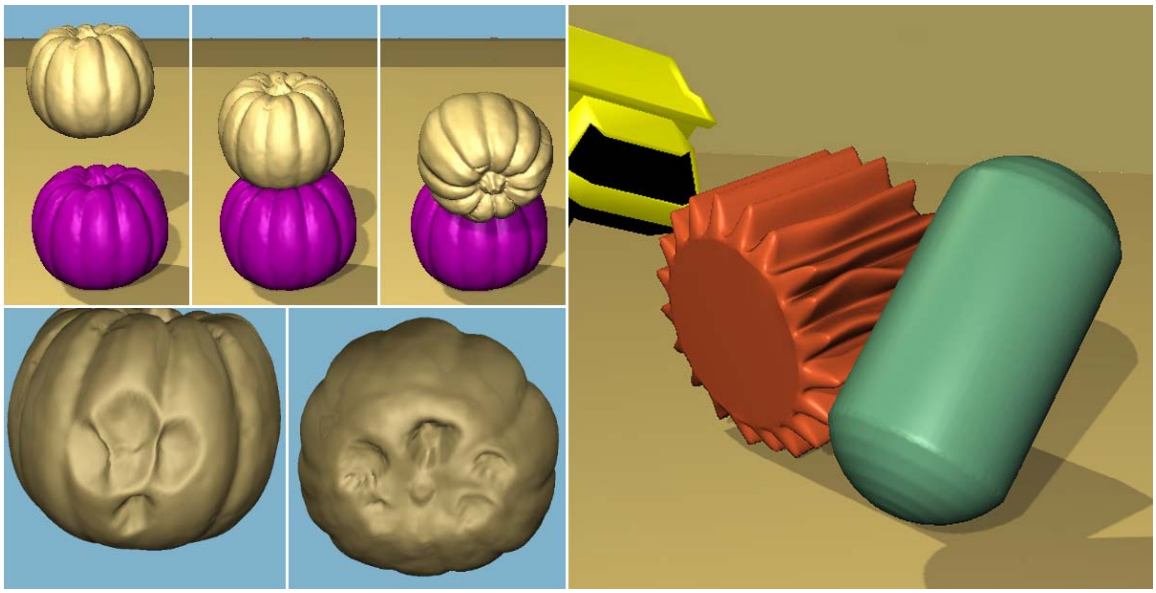


Figure 13. Rich Deformation of High-Resolution Geometry*. In the bottom-left corner, observe views from below of the top pumpkin as it collides with the bottom pumpkin and deforms.*

6.9 References

[EL00] EHMANN, S. A. AND LIN, M. C. 2000. Accelerated proximity queries between complex polyhedra by multi-level Voronoi marching. In Proceedings of International Conference on Intelligent Robots and Systems.

[GOM+06] GALOPPO, N., OTADUY, M. A., MECKLENBURG, P., GROSS, M., AND LIN, M. C. 2006. Fast simulation of deformable models in contact using dynamic deformation textures. In Proc. of ACM SIGGRAPH/Eurographics Symposium on Computer Animation, pp. 73–82.

[GV96] GOLUB, G. H. AND VAN LOAN, C. F. Matrix Computations. Johns Hopkins University Press, 3rd edition, 1996.

[HAR05] HARRIS, M. 2005. Mapping computational concepts to GPUs. In Matt Pharr, editor, GPU Gems 2, chapter 31. Addison Wesley, March 2005.

[OJSL04] OTADUY, M. A., JAIN, N., SUD, A., AND LIN, M. C. Haptic display of interaction between textured models. In Proc. of IEEE Visualization, 2004.

[PPG04] PAULY, M., PAI, D. K, AND GUIBAS, L. J. 2004. Quasi-rigid objects in contact. In Proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation.

[SHE94] SHEWCHUK, J. P. 1994. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Pittsburgh, PA, USA.

[STA03] STAM, J. 2003. Flow on surfaces of arbitrary topology. In Proceedings of ACM SIGGRAPH, pp. 724 – 731, San Diego, CA, USA, ACM Press.

[TL94] TURK, G. AND LEVOY, M. 1994. Zippered polygon meshes from range images. In SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques, pp. 311–318, New York, NY, USA, ACM Press.

[TW88] TERZOPOULOS, D. AND WITKIN, A. 1988. Physically based models with rigid and deformable components. IEEE Computer Graphics and Applications, 8(6).

```
void HighResRenderMesh::updateVBOfromTextures(
   FramebufferObject* fb,
   const TextureRef&  positionTexture,
   const TextureRef&  normalTexture)
{
  // read the vertex data back from framebuffer-attached texture
  // into the PBO
  if (!positionTexture.isNull())
  {
    fb->AttachTexture(GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D,
                      positionTexture->openGLID());
    glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
    fb->IsValid();
    debugAssertGLOk();
    glBindBuffer(GL_PIXEL_PACK_BUFFER_ARB, VBOs[POSITION]);
    debugAssertGLOk();
    glReadPixels(0, 0, pos_tex_height, pos_tex_width,
                 GL_RGBA /*BGRA*/, GL_FLOAT, 0);
    debugAssertGLOk();
  }

  // read the normal data back from framebuffer-attached texture
  // into the PBO
  if (!normalTexture.isNull())
  {
    fb->AttachTexture(GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D,
                      normalTexture->openGLID());
    glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
    fb->IsValid();
    debugAssertGLOk();
    glBindBuffer(GL_PIXEL_PACK_BUFFER_ARB, VBOs[NORMAL]);
    debugAssertGLOk();
    glReadPixels(0, 0, pos_tex_height, pos_tex_width,
                 GL_RGBA /*BGRA*/, GL_FLOAT, 0);
    debugAssertGLOk();
  }

  // Unbind
  glBindBuffer(GL_PIXEL_PACK_BUFFER_ARB, 0);
  debugAssertGLOk();
}
```

Listing 1. *Code Snippet 1. Routine to update two pixel buffers (PBO) from texture memory. The PBOs can then be interpreted as a vertex and normals buffer (VBO)*

```
/**
 * Compute sparse Kx product. Does *not* write alpha.
 *
 * @param x          x, an RGB(A) texture
 * @param y          y, result buffer, an RGB(A) buffer
 * @param tempbuffer z, tempbuffer, should match the format of y
 *                  (default RGBA)
 */
template <typename model_type> void compute_sparse_product(
      model_type&      model,
      texture_pointer* A,
      texture_pointer  x,
      texture_pointer  y,
      texture_pointer  tempbuffer)
{
   shared_ptr<GPUOps> gpu = model.m_gpu;
   shared_ptr<FramebufferObject> fbo = gpu->get_fbo();

   // Update the boundary information
   model.m_boundaryops->update_boundaries(model, x);
   DebugTexture(fbo, x);

   if (!tempbuffer)
      tempbuffer = gpu->m_tempbuffer2;

   // First pass, 3 neighbors and self
   tempbuffer->Attach(get_pointer(fbo), GL_COLOR_ATTACHMENT0_EXT);

   gpu->Ax1->SetTextureParameter("A00", A[0]->Texture());
   gpu->Ax1->SetTextureParameter("A01", A[1]->Texture());
   gpu->Ax1->SetTextureParameter("A02", A[2]->Texture());
   gpu->Ax1->SetTextureParameter("A20", A[3]->Texture());
   gpu->Ax1->SetTextureParameter("A21", A[4]->Texture());
   gpu->Ax1->SetTextureParameter("A22", A[5]->Texture());
   gpu->Ax1->SetTextureParameter("A30", A[6]->Texture());
   gpu->Ax1->SetTextureParameter("A31", A[7]->Texture());
   gpu->Ax1->SetTextureParameter("A32", A[8]->Texture());
   gpu->Ax1->SetTextureParameter("A80", A[18]->Texture());
   gpu->Ax1->SetTextureParameter("A81", A[19]->Texture());
   gpu->Ax1->SetTextureParameter("A82", A[20]->Texture());
   gpu->Ax1->SetTextureParameter("x", x->Texture());

   gpu->Ax1->SetMesh(model.GetParameterizedMesh());
   gpu->Ax1->Compute();
   tempbuffer->FastUnAttach();

   // Second pass, 3 neighbors and tempself
   y->Attach(get_pointer(fbo), GL_COLOR_ATTACHMENT0_EXT);
   gpu->Ax2->SetTextureParameter("A40", A[9]->Texture());
   gpu->Ax2->SetTextureParameter("A41", A[10]->Texture());
   gpu->Ax2->SetTextureParameter("A42", A[11]->Texture());
   gpu->Ax2->SetTextureParameter("A60", A[12]->Texture());
   gpu->Ax2->SetTextureParameter("A61", A[13]->Texture());
   gpu->Ax2->SetTextureParameter("A62", A[14]->Texture());
   gpu->Ax2->SetTextureParameter("A70", A[15]->Texture());
   gpu->Ax2->SetTextureParameter("A71", A[16]->Texture());
   gpu->Ax2->SetTextureParameter("A72", A[17]->Texture());
   gpu->Ax2->SetTextureParameter("x", x->Texture());
```

```
   gpu->Ax2->SetTextureParameter("tempy", tempbuffer->Texture());
   gpu->Ax2->SetMesh(model.GetParameterizedMesh());
   gpu->Ax2->Compute();
}
```

Listing 2. Code Snippet 2. *CPU driver code for sparse matrix multiply. Two passes on the GPU are invoked with the* `Compute()` *call.*

```
#define SAMPLER samplerRECT

float3 value3(SAMPLER sampler, float2 offset)
{ return texRECT(sampler, offset).xyz; }

float3 Ax(SAMPLER A0, SAMPLER A1, SAMPLER A2, float3 x, float2 coord)
{
   float3 y;
   y = mul( float3x3( value3(A0, coord),
                      value3(A1, coord),
                      value3(A2, coord)),
            x );
   return y;
}

void Ax1(
   in float2 coord : WPOS,
   uniform SAMPLER x,
   uniform SAMPLER A00, uniform SAMPLER A01, uniform SAMPLER A02,
   uniform SAMPLER A20, uniform SAMPLER A21, uniform SAMPLER A22,
   uniform SAMPLER A30, uniform SAMPLER A31, uniform SAMPLER A32,
   uniform SAMPLER A80, uniform SAMPLER A81, uniform SAMPLER A82,
   out float3 result : COLOR0)
{
   float3 x0 = value3(x, coord + float2(0.0, 1.0));
   float3 x2 = value3(x, coord + float2(1.0, 0.0));
   float3 x3 = value3(x, coord + float2(1.0, -1.0));
   float3 x8 = value3(x, coord);
   result  = Ax(A00, A01, A02, x0, coord);
   result += Ax(A20, A21, A22, x2, coord);
   result += Ax(A30, A31, A32, x3, coord);
   result += Ax(A80, A81, A82, x8, coord);
}
```

Listing 3. Fragment Program 1*. Compute* $\mathbf{Ax} = \mathbf{A}_l\mathbf{x}+\mathbf{A}_r\mathbf{x}$ *with D2T mapped sparse matrix in two passes. The intermediary result from* `Ax1()` *is passed on to* `Ax2()` *as input in the second pass.*

```
void Ax2(
   in float2 coord : WPOS,
   uniform SAMPLER x, uniform SAMPLER tempy,
   uniform SAMPLER A40, uniform SAMPLER A41, uniform SAMPLER A42,
   uniform SAMPLER A60, uniform SAMPLER A61, uniform SAMPLER A62,
   uniform SAMPLER A70, uniform SAMPLER A71, uniform SAMPLER A72,
   out float3 result : COLOR0)
{
   float3 x4 = value3(x, coord + float2(0.0, -1.0));
   float3 x6 = value3(x, coord + float2(-1.0, 0.0));
   float3 x7 = value3(x, coord + float2(-1.0, 1.0));
   result  = value3(tempy, coord);
   result += Ax(A40, A41, A42, x4, coord);
   result += Ax(A60, A61, A62, x6, coord);
   result += Ax(A70, A71, A72, x7, coord);
}
```

Listing 4. ***Fragment Program 2.*** *Compute* $\mathbf{Ax = y+A_rx}$ *with D2T mapped. The intermediary result from* `Ax1()` *is passed as input* `Ax2()`.

```
void main(
   float4 pos : POSITION,
   in float4 tin : TEXCOORD0,
   out float4 eyepos : TEXCOORD0,
   out float4 tidpos : TEXCOORD1,
   out float4 clippos : POSITION
         )
{
   eyepos = mul(glstate.matrix.modelview[0], pos);
   tidpos = tin;
   clippos = mul(glstate.matrix.mvp, pos);
}
```

Listing 5. ***Vertex Program 1.*** *Transform position to eye space.*

```
void main(
   float4 pos : WPOS,
   float4 eyepos : TEXCOORD0,
   float4 tidpos : TEXCOORD1,
   uniform samplerRECT triangleidmap : TEX0,
   out float3 result : COLOR0
         )
{
   // Copy the triangle ID to green
   result.g = value(triangleidmap, tidpos.xy);

   // transfer depth (with and without perspective divide)
   // z component of eye space position is distance to the eye
   result.rb = eyepos.zw;
}
```

Listing 6. ***Fragment Program 3****. Rasterize distance to eye.*

```
void main(
   in float2 coord : WPOS,
   uniform samplerRECT texture1,
   uniform samplerRECT texture2)
{
   // Subtract texture values and copy to red
   float2 val1 = f2texRECT(texture1, coord.xy);
   float2 val2 = f2texRECT(texture2, coord.xy);
   result.r = val1.r - val2.r;

   //Copy triangle ID to green and blue
   result.g = val1.g;
   result.b = val2.g;
}
```

Listing 7. ***Fragment Program 4****. Compute per-texel depth differences.*

```
void tagcontactobj1( // code for object 1
   in float2 coord : WPOS, in float2 texcoord : TEXCOORD0,
   uniform float3 lowresnormal,
   uniform samplerRECT pdtexture, uniform samplerRECT trianglemap,
   out TYPE result : COLOR0)
{
   float3 pd = value3(pdtexture, texcoord);
   float triangle_id = value(trianglemap, coord);
   result = 0.0;
   // compare triangle ID and penetration depth.
   // Note: the triangle ID for object 1 is stored
   // in the green component of pd
   if ((pd.r > 0.0) && (abs(pd.g - triangle_id) < 0.00001))
   {
      //store inwards lowres normal
      result.xyz = lowresnormal;
      //store penetration depth
      result.a = pd.x;
   }
   else { discard; }
}

void tagcontactobj2( // code for object 2
   in float2 coord : WPOS, in float2 texcoord : TEXCOORD0,
   uniform float3 lowresnormal,
   uniform samplerRECT pdtexture, uniform samplerRECT trianglemap,
   out TYPE result : COLOR0)
{
   float3 pd = value3(pdtexture, texcoord);
   float triangle_id = value(trianglemap, coord);
   result = 0.0;
   // compare triangle ID and penetration depth.
   // Note: the triangle ID for object 2 is stored
   // in the blue component of pd
   if ((pd.r > 0.0) && (abs(pd.b - triangle_id) < 0.00001))
   {
      //store inwards lowres normal
      result.xyz = lowresnormal;
      //store penetration depth
      result.a = pd.x;
   }
   else { discard; }
}
```

Listing 8. Fragment Program 5. *Tag colliding texels in the D2T by transferring the collision data from D with the appropriate mapping and with triangle checking.*

```
void ContactComputePolicy::ComputePolicy(const Matrix4 & contactCamMVP)
{
   // load contact camera matrices
   glMatrixMode(GL_TEXTURE);
   static Matrix4 bias(
      0.5f, 0.0f, 0.0f, 0.5f,
      0.0f, 0.5f, 0.0f, 0.5f,
      0.0f, 0.0f, 0.5f, 0.5f - .000001f,
      0.0f, 0.0f, 0.0f, 1.0f);
   glLoadMatrix(m_bias);
   glMultMatrix(contactCamMVP);
   CheckErrorsGL("Loaded contact camera matrices");

   // Render into D2T atlas
   m_mesh->RenderNearContactToAtlas(contact->Point(m_numobj), m_normal);
}
```

Listing 9. ***Code Snippet 3.*** *Set up texture matrix for projection of contact domain to D2T atlas and render into D2T atlas.*

```
void generate_normals(
   in float2 coord : WPOS,
   uniform samplerRECT bodypos,
   out float3 normal : COLOR0
                     )
{
   // fetch body position from position texture
   float3 pos       = value3(bodypos, coord);
   float3 up        = value3(bodypos, coord + float2(0,1)) - pos;
   float3 down      = value3(bodypos, coord + float2(0,-1)) - pos;
   float3 left      = value3(bodypos, coord + float2(-1,0)) - pos;
   float3 right     = value3(bodypos, coord + float2(1,0)) - pos;
   float3 upright   = value3(bodypos, coord + float2(1,1)) - pos;
   float3 downright = value3(bodypos, coord + float2(1,-1)) - pos;
   float3 upleft    = value3(bodypos, coord + float2(-1,1)) - pos;
   float3 downleft  = value3(bodypos, coord + float2(-1,-1)) - pos;
   float3 norm      = (float3)0;

   norm += normalize(cross(up, left));
   norm += normalize(cross(left, down));
   norm += normalize(cross(down, right));
   norm += normalize(cross(right, up));
   norm += normalize(cross(upright, upleft));
   norm += normalize(cross(upleft, downleft));
   norm += normalize(cross(downleft, downright));
   norm += normalize(cross(downright, upright));

   normalize(norm);
   normal = norm;
}
```

Listing 10. ***Fragment Program 6.*** *Generate normal map by sampling of each D2T texel neighborhood.*

Chapter 7

Real-Time Particle Systems on the GPU in Dynamic Environments

Shannon Drone[13]
Microsoft Corporation

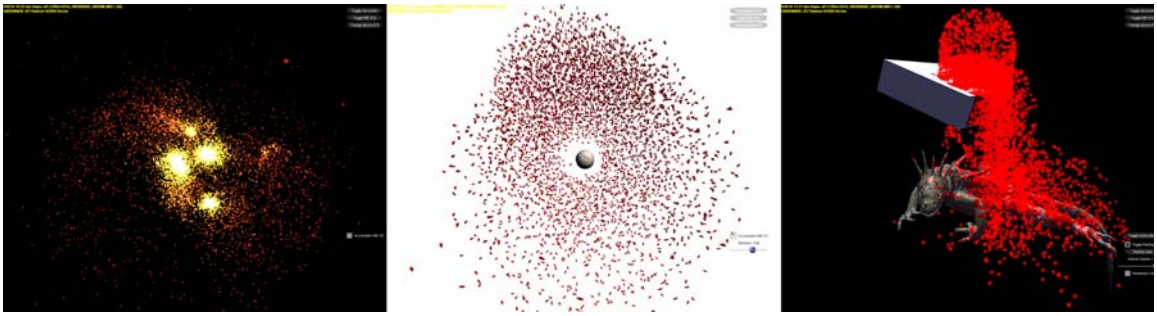


Figure 1. *From left to right, an N-Body Gravity simulation, a flocking simulation, and particles interacting with and influencing their environment.*

7.1 Introduction

Particle systems ([Reeves83, Sims90, McAllister00]) have been the mainstay of video game effects for the past decade. They have been used to simulate everything from explosions ([Burg2000]) to swarms of insects ([Reynolds87]). As more and more processing power is becoming available on commodity graphics processors, many video game subsystems are now moving over to the GPU. Particle systems have moved with them, but in doing so, have lost some of their functionality in the move.

In this chapter we introduce several methods for creating advanced interaction particle system simulations whose data and computations reside entirely on the GPU. We use non-parametric particle systems on the GPU to display complex particle behavior otherwise reserved for CPU based particle systems. In this chapter we cover the basics of non-parametric particle systems, particle-to-particle interactions, and particle versus scene interactions.

[13] email: shanond@microsoft.com

For the intents of this chapter, we base our approach on an assumption that the particle system data is not instrumental to gameplay and that the CPU does not need to results of particle system operations to perform any other game-related functions. However, it is a fairly straightforward extension of our approach to provide particle data back to the CPU by read-back.

7.2 Rendering System Requirements

While many of the methods described here can be adapted to work on the majority of consumer video graphics hardware currently in the market, some techniques require the use of more advanced features that can only be found on Direct3D 10-level graphics devices (as described in [Blythe06]).

For the following techniques we assume that the video hardware is a recent video card that supports at least a Direct3D 10 level of functionality. In our case we are specifically going to take advantage of such features of this generation of hardware as additive alpha blending; instancing support; the ability render directly to volume textures; the ability to sample textures or data buffers from any stage of the pipeline; support for pixel, vertex, and geometry shaders; the ability to save transformed geometry back into GPU memory; texture array support; and automatic generation of mip-maps.

7.3 Non-Parametric Particle Systems

Parametric or stateless particle systems are easy to handle in programmable graphics pipelines. Because each particle position is described parametrically the position of the particle at any time can be determined by plugging that time into an equation of motion. This approach has two main benefits. The first is that it requires no extra storage for intermediate particle state. The second is that it is an exact analytical solution to the path of motion for the particle. No integration of the equations of motion is required to find the position of the particle.

Unfortunately, there are drawbacks to using parametric systems. The main one is that once set, the motion of a particle cannot change. This limits the ability of a parametric particle system to react to its environment in real-time. In addition, it limits the system to paths of motion with known analytical solutions (as described in [Lutz04]).

For our work, we use non-parametric particle systems similar to [Lutz04]. These work on the premise that the equations of acceleration are integrated over the course of the simulation to compute instantaneous velocity. The velocity equation is integrated over the course of the simulation to compute instantaneous position. This approach is less accurate than a purely analytical parametric solution, but maintains a level of flexibility and interaction far beyond a parametric system.

7.3.1 Storage Requirement

In order to integrate the equations of acceleration and velocity, we must store the immediate values for the previous frame's instantaneous velocity and position. These will be known as the particle's state. For the remaining techniques, we can store particle state using either of two storage objects readily available on current graphics hardware.

The first option is to store state in a vertex buffer. In this approach, each vertex represents the state of one particle in its entirety. It must contain at a minimum, the instantaneous position and velocity of the particle at the current time value. The particles are stored linearly in the vertex buffer object.

The second option is to store the particle state in series of floating point textures. Whereas [Lutz04] used several individual textures to store the data, we split the data between multiple slices of a single texture array. A texture array is a single object that acts as a container for an array of traditional textures. The first array slice stores instantaneous position and the second instantaneous velocity. Additional array slices may be used to store additional data. This data could be stored in one-dimensional textures, but size limitations on one-dimensional textures for current API and hardware versions would limit us to 8192 particles in the best case. Therefore, we store particle state in two-dimensional textures where the height and width of the texture are the next largest integral square of the number of particles.

7.3.2 Integrating the Equations of Motion

Because the particle state is integrated using a series of instantaneous accelerations and velocities, the accuracy of the solution depends entirely on the length of time between the calculation of the previous values and the current values as well as the integration technique used. Simple Euler integration will work in most cases where the behavior is simple or where the time between calculations is sufficiently small. However, a more advanced integration such as a Runge-Kutta based integration scheme maybe be used where further accuracy is required. Note that using a more advanced integration solution may require storage of several previous particle states. For the techniques expressed here, we use Euler integration.

7.3.3 Saving Particle States

The current methodology of integrating particle motion requires a read-modify-write operation on the particle state data. The Euler integration scheme for velocity requires that the current velocity be known and added to the instantaneous acceleration scaled by the current time step. Unfortunately, read-modify-write operations are illegal in the programmable parts of the current graphics pipeline (they are allowed in the blend stages which are currently not programmable). The solution is use a "ping-pong" technique to essentially double buffer the data. In the particle update phase, the buffer

or texture being sampled contains the instantaneous particle state for the previous frame. The particle update phase stores the new instantaneous particle state in other buffer or texture. The buffers or textures are swapped for the next frame so that the particle update phase is always reading from the previous frame's data.

7.3.4 Changing Behaviors

Because our particles are no longer affixed to a predestined path of motion, changing behaviors of individual particles is as easy as changing their individual velocities or positions. While these will result in an immediate change of motion for the particle, a change in position will cause a break in the C^1 continuity (or the position curve), while a change in velocity will cause a break in the C^2 continuity (i..e the derivative of the position curve). In the following techniques, we will only change acceleration, and therefore only break C^3 of the position curve. This results in a much smoother visual appearance of particle motion.

7.4 Particles That React to Other Particles

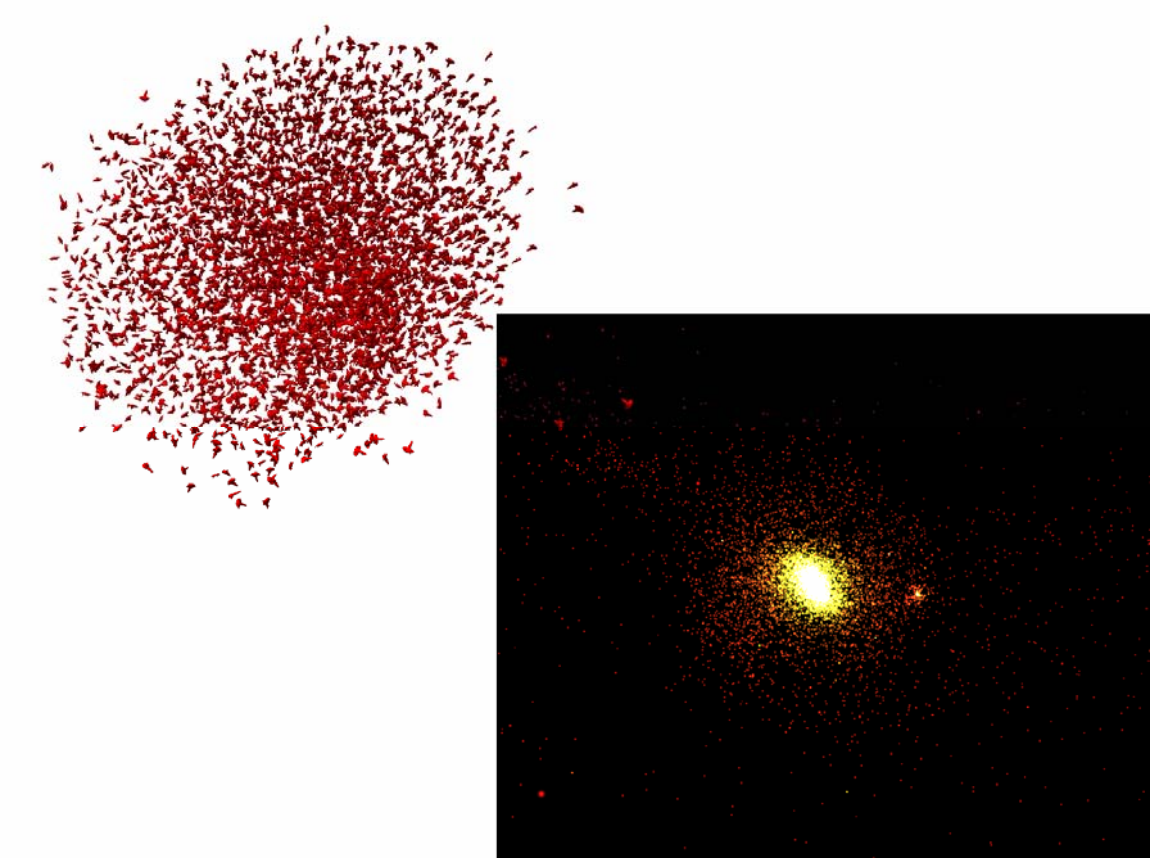


Figure 2. Flocking and gravity simulations

7.4.1 N-Body Problems

Many particle systems require that every particle influence every other particle in the system. These are generally classified as N-Body problems. We outline a method of dealing with N-Body problems on the GPU.

7.4.2 Force Splatting for N^2 Particle Interactions

The goal of *force splatting* is to project the force from one particle onto all other particles during a single operation. In this case, the operation is the rendering of a quad primitive. We create a texture that acts and an accumulation buffer for all forces applied to the particles. This buffer will be the target of the rasterization operations that will accumulate particle forces. Each texel in the force texture holds the accumulated forces acting upon a single particle. We also create a stack of *N* quad primitives, where N is the number of particles in the system. The dimensions of the quads are such that they will exactly cover the force buffer when rasterized. The four vertices of each quad in the stack contain a vertex element which identifies the exact particle represented by the quad. During rasterization, this interpolated vertex element is used to fetch properties of the particle from the particle texture or the particle buffer.

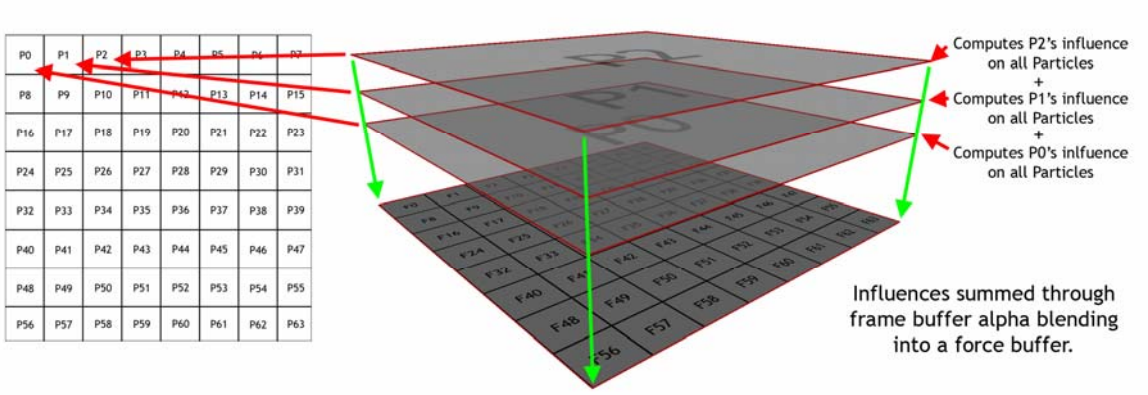


Figure 3. *Force splatting by rendering multiple into a force texture with alpha blending*

During the rasterization of a single quad, the forces are calculated between the particle being rasterized to and the particle represented by the vertex element in the vertices of the quad. Forces are accumulated by rendering successive quad with additive alpha blending enabled.

While less than elegant in terms of algorithmic complexity, the force splatting algorithm exploits the fast rasterization and alpha blending capabilities of modern graphics hardware without the need to continually recreate complex space partitioning structures on the GPU.

7.4.3 Gravity Simulation

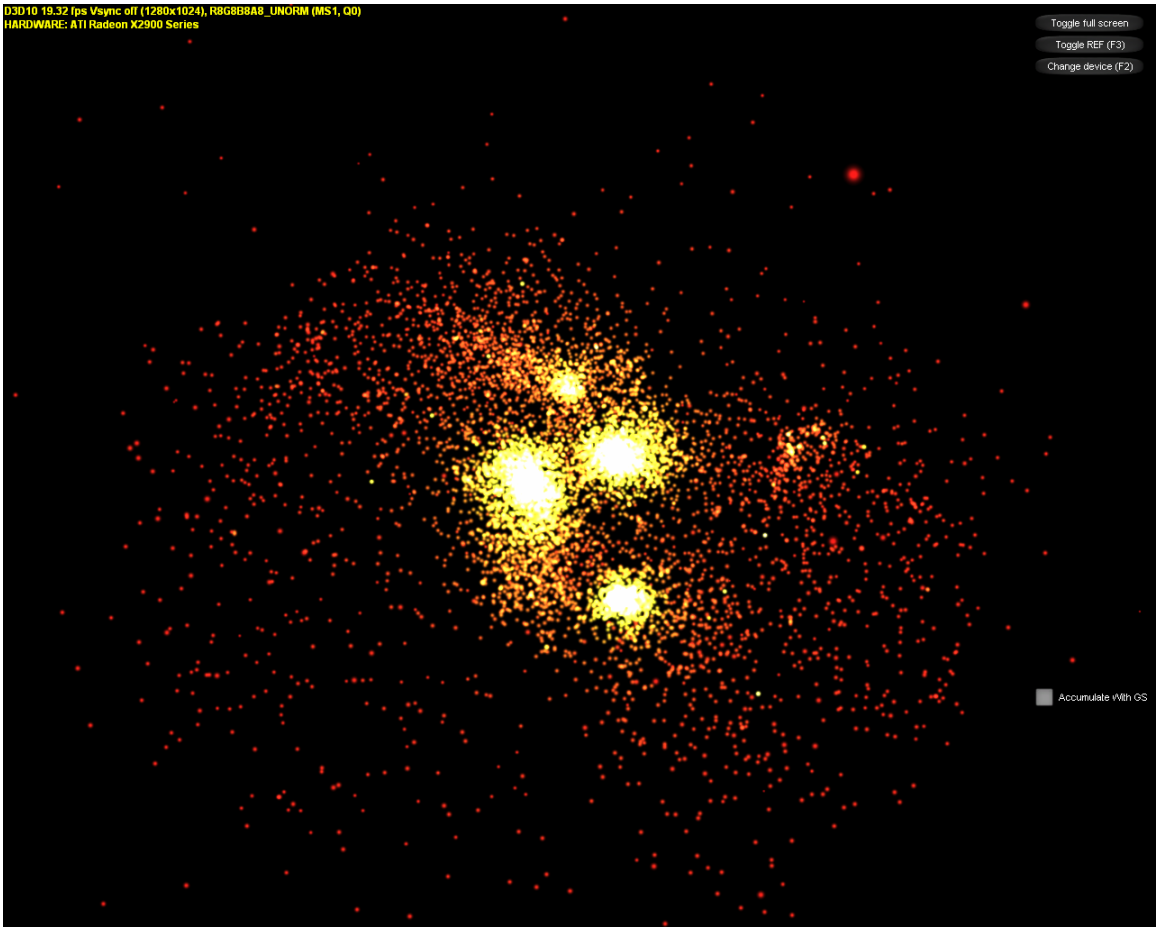


Figure 4. *N-body gravity simulation using force splatting to accumulate forces between all N particles*

7.4.3.1 Using Force Splatting for Gravity Interactions

To compute the gravitational force of all particles to all other particles, we use the method of force splatting mentioned above to accumulate all of the forces imparted on each particle in the system. In the particle update phase, this force is divided by the particle's mass to determine the instantaneous acceleration of the particle. The equations of motion are integrated, and the particle system is updated.

7.4.4 Flocking Particles on the GPU

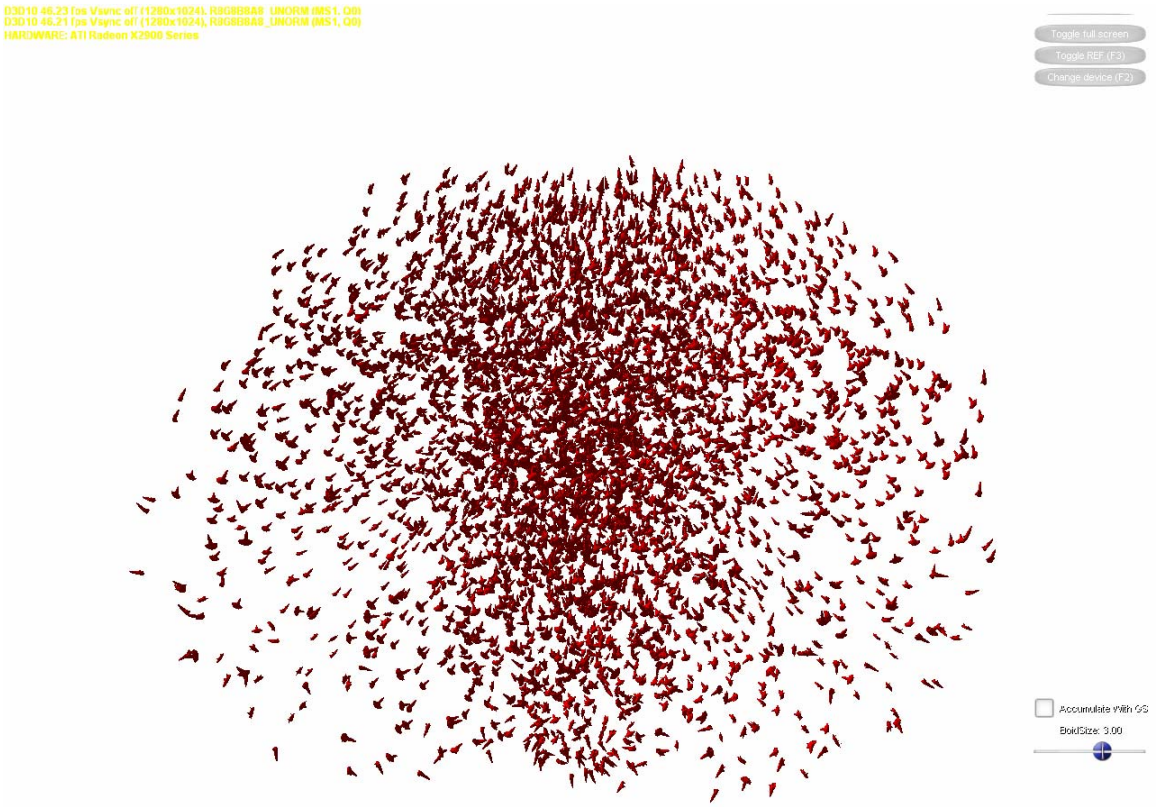


Figure 5*. A boids implementation handled entirely on the GPU. Particles use force splatting for collision avoidance, and separation while using fast mip map generation for coherence and goal seeking. A single space-ship mesh is then instanced using particle position and orientation as a transform.*

Perhaps more relevant to game development is the idea of flocking particle systems. Oftentimes particle systems are used to create the illusion of flocks of birds or bugs swarming around a light or fallen comrade. Traditional flocking behaviors need to follow a few simple rules in order to look plausible. In this situation, the rules are collision avoidance, separation, cohesion, and alignment. See [Reynolds87, Reynolds99] for in-depth descriptions of flocking behaviors.

7.4.4.1 Force Splatting for Collision Avoidance and Separation

The flocking simulation takes advantage of the previous N^2 force splatting to avoid collisions between particles as well as to maintain a certain comfortable separation between all particles. Instead of computing the gravitational attraction between particles,

we're computing a repellant force for each particle based upon either how close the particles are to colliding or how much space is between particles.

7.4.4.2 Fast Averaging for Cohesion and Alignment

Behaviors such as cohesion and alignment rely on the knowledge of the average position and average velocity of the particles respectively. Fortunately, modern graphics hardware provides a fast way of averaging entire textures by being able to generate mip-maps on the fly. By sampling from the smallest mip-level during the particle update phase, we can create a force vector from the particle to the center of mass for cohesion or create a force vector that aligns our particle with the average velocity of all other particles. This force vector is added to the force vector sampled from the force accumulation texture.

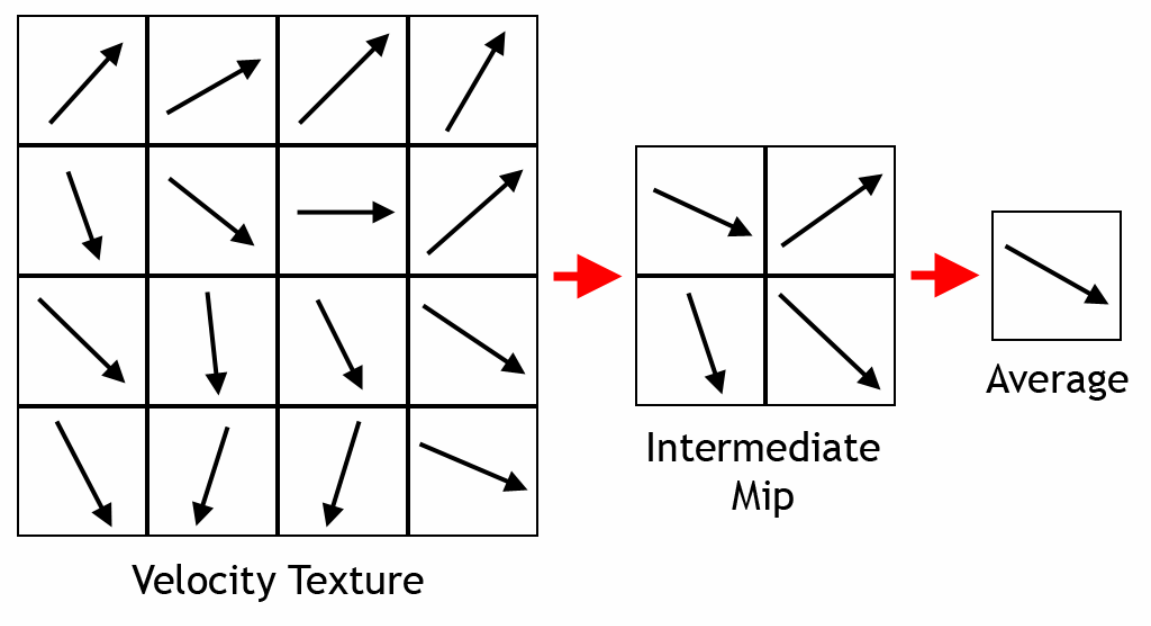


Figure 6*. Fast averaging of particle states by generating mip-maps*

7.5 Particles Reacting to Their Environments

In order for non-parametric particle systems to have a true advantage over parametric or scripted systems, they must react to their environments as well as to each other.

7.5.1 Reacting to Spherical Objects

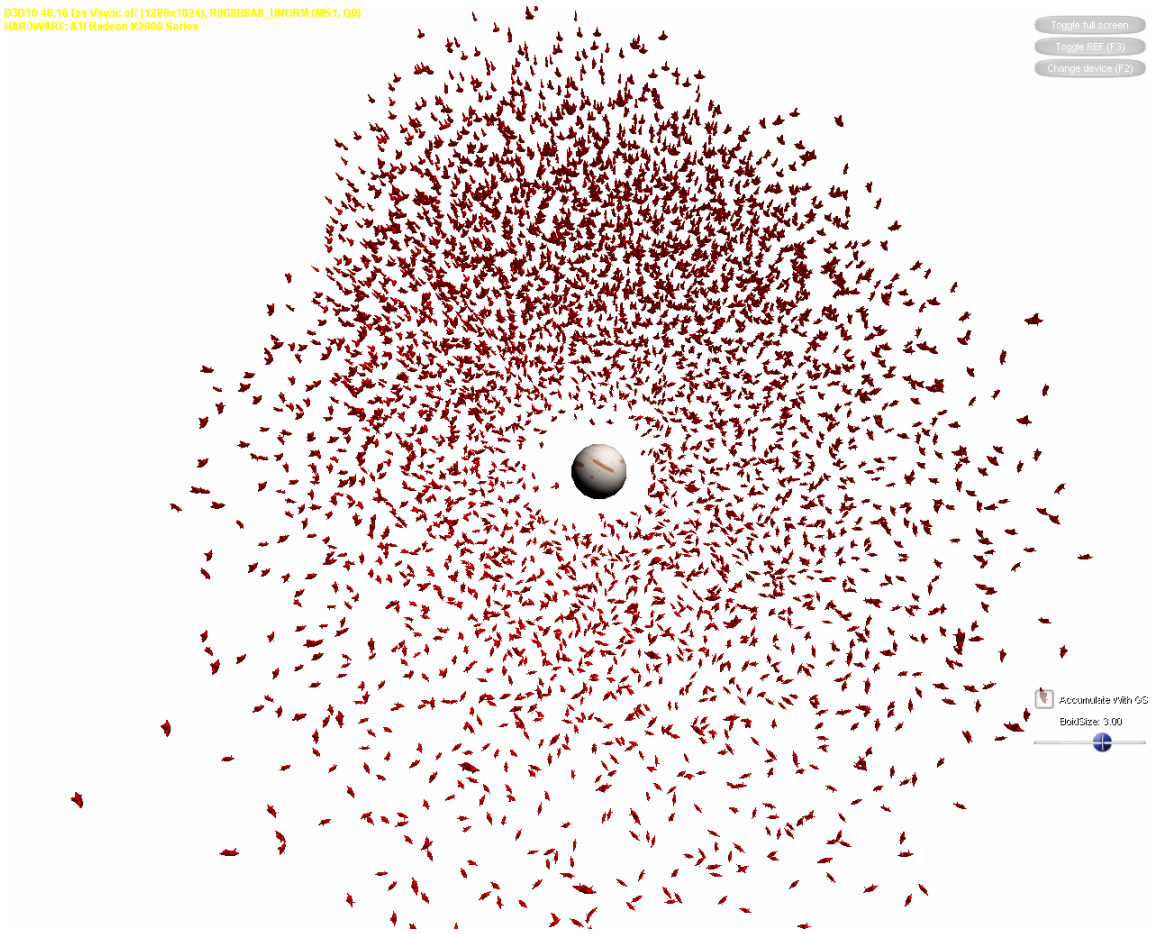


Figure 7. *Thousands of spaceships fleeing from a user-controlled obstacle*

The simplest way to interact with a particle system is to influence it through a limited set of "point charges." We use this approach for flee and seek behavior. To repel or attract an entire flock, we create a limited set of spherical targets and pass in their parameters as shader variables. This allows the particles to react to "point charges" introduced into the system. The 'seek and flee' algorithms are a straight GPU implementation of [Reynolds99].

7.5.2 Reacting to Arbitrary Objects Using Render-to-Volume

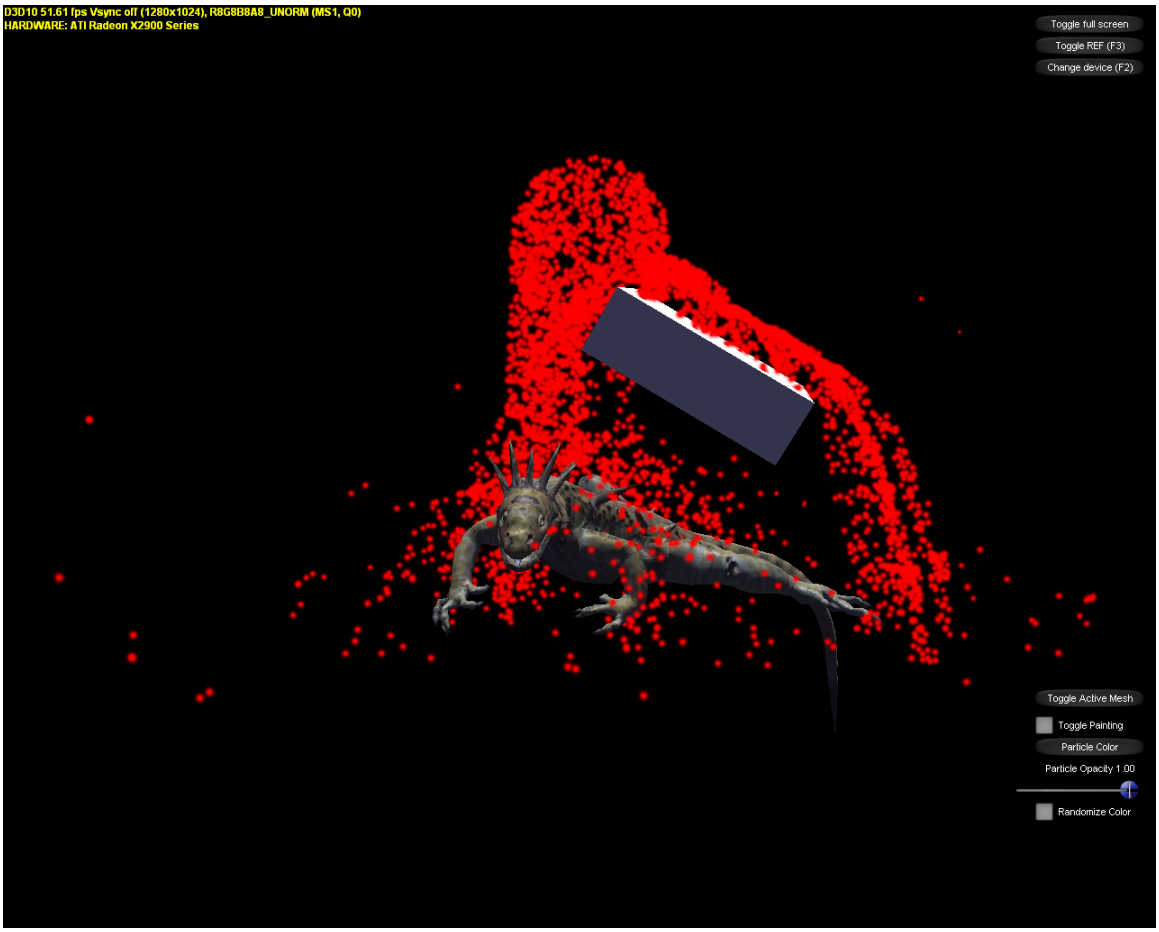


***Figure 8**. Particles bounce off and flow along both the box and animated lizard*

Many times particles must interact with shapes that cannot be accurately described by a fixed number of spheres. [Lutz04] partitioned spaced into a two-dimensional grid. This effectively limited the problem of collision to a height-field. In our algorithm we partition the space in which the particles will interact into a regular three-dimensional grid. Before the particle update phase, the scene geometry is placed into this grid in such a way that each cell in the grid contains the plane equation and velocity of the scene geometry that intersects that grid cell.

During the particle update phase, the particles determine which grid cell they are in and fetch the plane equation and velocity from the grid cell. These are used to determine whether there has been an intersection with the scene geometry and the new position and velocity of the particle if such a collision occurred.

This method requires that two problems be overcome. The first is how to efficiently populate the three-dimensional grid with scene data. The second is how to efficiently fetch this data during the particle update phase. Fortunately, both problems have the same solution. Modern hardware provides support for regular three-dimensional grid

structures in the form of volume textures. Additionally, volume textures can be rendered into or sampled using the graphics hardware.

7.5.2.1 Populating the Volume Texture

The volume texture must be populated with the scene geometry once slice at a time. Normally this would require a separate invocation of the rendering pipeline for each slice of the volume and then again for each object to be rendered. However, the latest advances in graphics hardware provide the ability to bind all slices of the volume to the pipeline at once and selectively output geometry to each slice, therefore reducing the process to one invocation of the rendering pipeline for each object. This latest advancement in graphics hardware comes in the form of a new addition to the rendering pipeline called the *geometry shader*. In addition to being able to specify output slices into a volume render target, the geometry shader can also perform operations on whole primitives.

The process works as follows: the scene geometry is drawn with hardware instancing turned on. We draw *S* instances of the scene geometry where *S* is the number of slices of the volume texture. In the shader, each triangle primitive is sent to a different slice of the volume depending on the instance ID of the geometry.

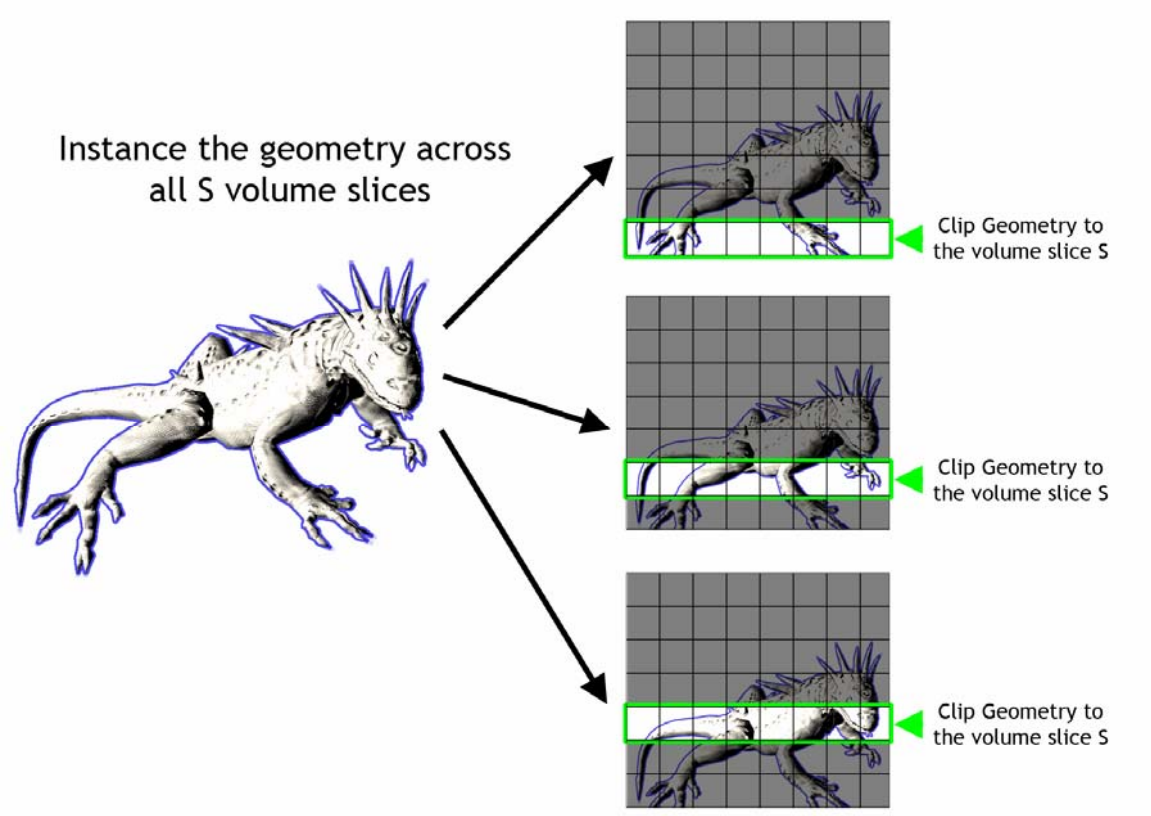


Figure 9*. Rendering an object into a volume using instancing to send it to all slices*

Using the aforementioned geometry shader, the plane equation for the primitive is computed and passed along to the pixel shader along with the velocities of each of the vertices. In order to ensure only geometry that passes through a particular slice ends up being rasterized to that slice, user specified clip planes are provided to clip any geometry that falls outside of its specified slice. The pixel shader then outputs the plane equation and interpolated velocity into the volume texture.

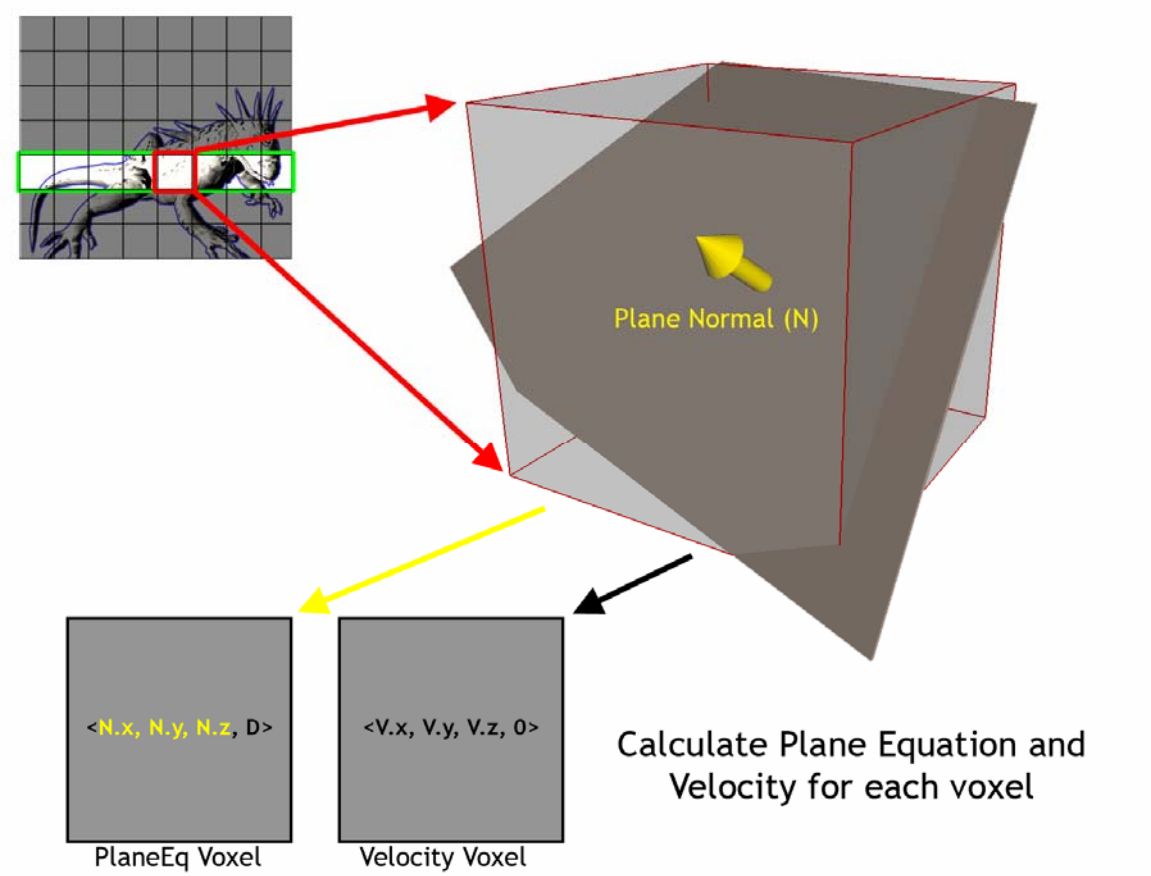


Figure 10*. The plane equation and velocity are rendered into each voxel of the volume*

7.5.2.2 Sampling the Volume Texture

In the particle update phase, the particle volume texel that encompasses the particle is sampled for its plane equation and velocity. The particle is then checked for collisions against the plane equation. If a collision occurs, the particle is deflected according to its own velocity, the plane equation, and the plane velocity.

7.5.2.3 Resolving Aliasing

With detailed geometry or a coarse volume texture representation, multiple primitives may be rasterized into the same volume cell. To store all plane equations and velocities that intersect that grid cell would take too much video memory and require multiple fetches in the sampling phase. Therefore, we keep only the most important plane equation and velocity to use in our computations. We do this by rendering the scene geometry into the volume texture from the direction that the majority of the particles will be traveling in. This is often the point of view of the emitter. We then use the depth test in the hardware to ensure that the primitive closest to the camera position used when rendering the scene into the volume will be kept. Since the majority of the particles are moving in the direction away from the camera we can ensure that in an ideal situation most particles would hit this plane before hitting any other plane that would also occupy this particular cell. However, the incorrect results may be achieved for particles traveling in a direction that is too different from the average direction. This error can also be avoided with a denser volume texture.

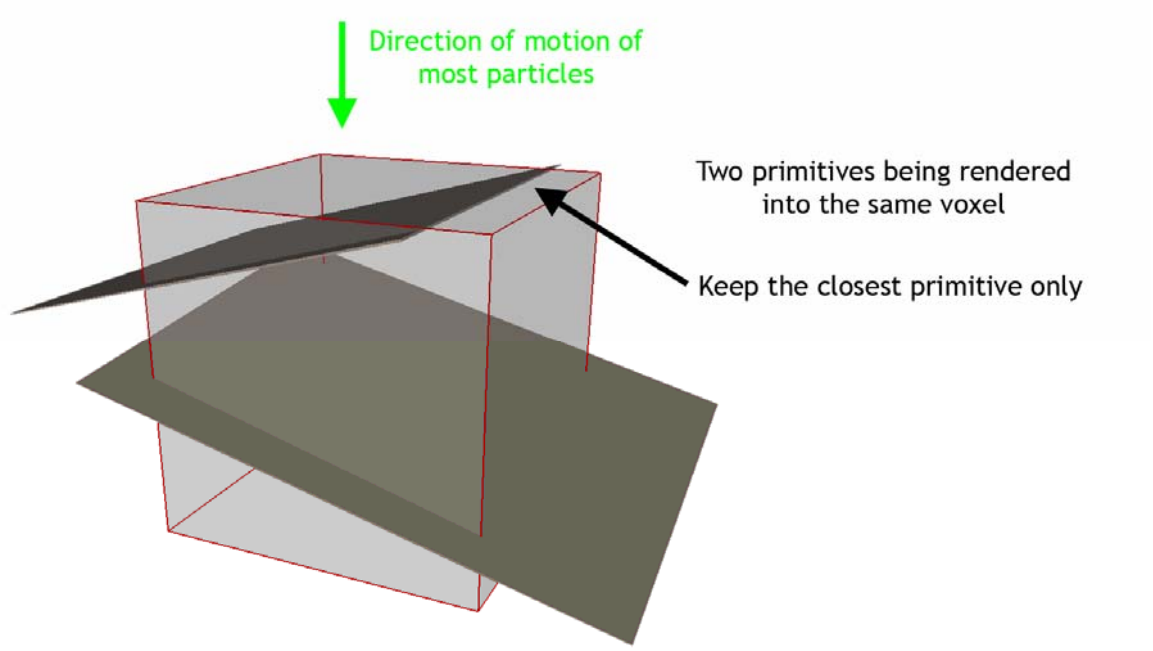


Figure 11. *Aliasing can occur when two primitives occupy the same voxel. Keep the one closest to the direction of motion of most particles.*

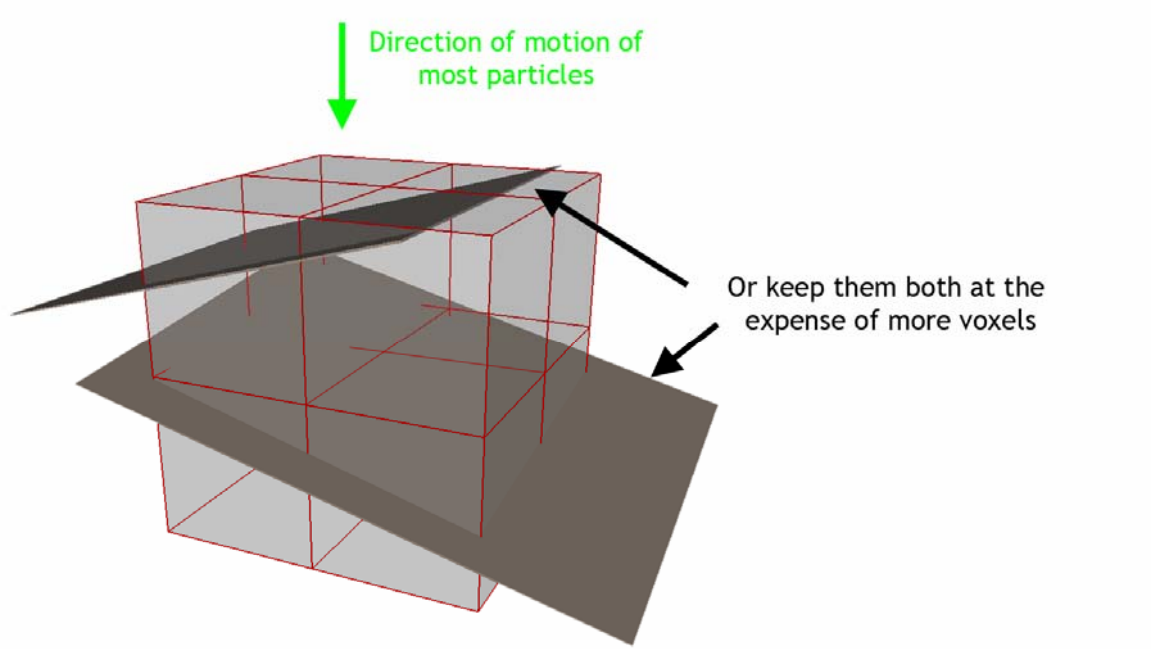


***Figure 12**. One way to combat aliasing is to use a denser volume texture*

7.6 Environments That React to Particles

***Figure 13**. The particles paint into the diffuse channel of the box and lizard when they intersect the objects.*

Finally, we show how particles can actually affect their environments. We use the particles to affect the appearance of the world geometry.

7.6.1 Painting with Particles Using a Gather Approach

Finally, we show how the appearance of the scene geometry can change based upon it's interaction with particles. In particular, the particles will apply paint to any part of the object that they encounter.

7.6.1.1 Rendering the Position Buffer

First we need to create a position buffer for each object in the scene. The position buffer is a floating point texture that contains a world-space position for each texel in the object's UV space. This is effectively a UV to world space mapping. To populate the position buffer, we render the mesh using the texture uv coordinates as position coordinates. This renders the mesh geometry in UV space. The pixel shader then outputs the interpolated position data into the position texture. Care must be taken to ensure that the *uv* element being used is a unique parameterization of the mesh, otherwise the results will be incorrect.

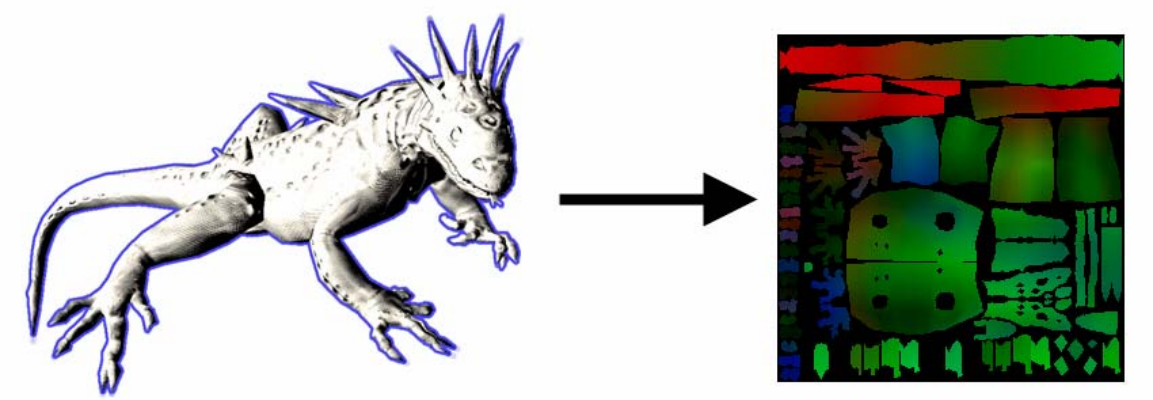


Figure 14*. Creation of the position texture: World position is rendered into UV space.*

7.6.1.2 Gathering Paint Splotches

With the position buffer populated, we need to gather particles from the particle buffer or texture and determine whether they intersect the mesh. If so, we add their paint to a paint texture. We handle this by setting the paint texture as a render target and rasterizing a quad that, when rendered, covers the render target exactly. During rasterization, we sample the world-space position from the position texture for the current texel. We then iterate over the particles in the particle buffer or texture. For each particle, we determine if it is close enough to the world-space position in the

position buffer to leave any paint. If so, we add the paint influence to the total paint output for this pixel shader invocation.

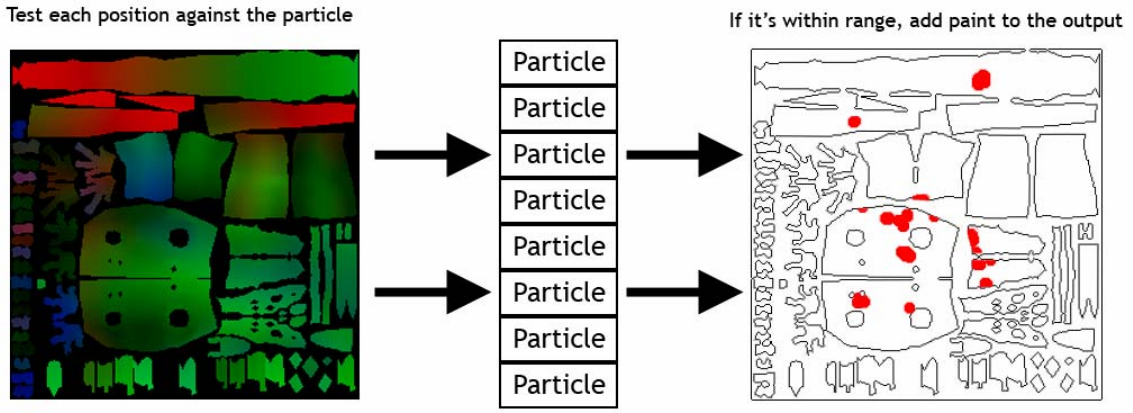


Figure 15*. A pixel shader passes over the position texture. For each particle, it determines whether the current position intersects the particle. If it does, it outputs an appropriate amount of paint into the diffuse texture.*

7.6.1.3 Amortizing the Gather over Time

For systems containing thousands of particles, iterating over all particles during gather time may not provide the best frame rate. For hardware with a fixed instruction count it may not be possible to loop over all particles. We amortize the cost of gathering over several frames by determining a fixed amount of particles to gather. For example, for the first frame we gather the first G particles. For the next frame we gather the next G particles, and so on until we loop back around to the beginning of the particle buffer. This gives much better performance with little loss in the quality of the effect.

7.7 Acknowledgements

We would like to thank Matt Dudley for the lizard art.

7.8 References

[BLYTHE06] BLYTHE, D. 2006. The Direct3D 10 system. ACM Trans. Graph. 25, 3, pp. 724-734.

[BURG00] J. VAN DER BURG. 2000. Building an Advanced Particle System. Gamasutra, June 2000.

[LUTZ04] LUTZ, L. 2004. Bulding a Million Particle System. In proceedings of Game Developers Conference, San Francisco, CA, March 2004.

[MCALLISTER00] McAllister, D. K. 2000. The Design of an API for Particle Systems. University of North Carolina Technical Report TR 00-007

[REEVES83] REEVES, W. T. 1983. Particle systems -- a technique for modeling a class of fuzzy objects. ACM Transactions on Graphics, 2(2), pp. 91-108, Apr. 1983.

[REYNOLDS87] REYNOLDS, C. 1987. Flocks, Herds and Schools: A Distributed Behavioural Model. Computer Graphics, 21(4), pp.25-34.

[REYNOLDS99] REYNOLDS, C. W. 1999. Steering Behaviors for Autonomous Characters, in the proceedings of Game Developers Conference 1999 held in San Jose, California. Miller Freeman Game Group, San Francisco, California. pp. 763-782.

[SIMS90] SIMS, K. 1990. Particle Animation and Rendering Using Data Parallel Computation. ACM Computer Graphics (SIGGRAPH '90), 24(4), pp. 405-413, August 1990.

Chapter 8

Finding Next Gen – CryEngine 2

Martin Mittring[14]
Crytek GmbH

Figure 1. *A screenshot from the award-winning* Far Cry *game, which represented "next gen" at the time of its release*

Figure 2. *A screenshot from the upcoming game* Crysis *from Crytek*

[14] email: martin@crytek.de

8.1 Abstract

In this chapter we do not present one specific algorithm; instead we try to describe the approaches the German company named Crytek took to find certain rendering algorithms that work well together. We believe this information is valuable for anyone that wants to implement similar rendering algorithms because often the implementation challenges arise when combining with other algorithms. We will also describe briefly the path to it as that covers alternative approaches you also might want to consider. This is not a complete description of everything that was done on the rendering side because for this chapter we picked certain areas that are of interest specifically for this audience and limited ourselves to a presentable extend.

The work presented here takes significant advantage of research done by the graphics community in recent years and combines it with novel ideas developed within Crytek to realize implementations that efficiently map onto graphics hardware.

8.2 Introduction

Crytek Studios developed a technically outstanding *Far Cry* first person shooter game and it was an instant success upon its release. *Far Cry* raised the bar for all games of its genre. After our company shipped *Far Cry*[1]*,* one convenient possibility was to develop a sequel using the existing engine with little modifications - more or less the same engine we used for *Far Cry*. While this could have been an easy and lucrative decision, we believed that it would prove to be limiting for our goals – technically and artistically. We made the decision that we want to develop a new next-generation engine and improve the design and architecture, along with adding many new features. The new game, named *Crysis*[2], would follow *Far Cry* with the same genre, but would tremendously increase in scope – everything had to be bigger and better. The new engine, the *CryEngine 2*, would make that possible.

After reading the design document and an intense deliberation session amongst all designers, programmers and artists, we arrived at a set of goals for the new engine to solve:

[1] Shipped March 2003, Publisher: Ubisoft, Platform: PC
[2] Not released yet, Publisher: Electronic Arts, Platform: PC

- **The game would contain three different environments**

Many objects, height map, ocean, big view distance, ambient lighting with one main directional light source

Figure 3. *Jungle paradise*

Many point lights, dark, huge room like sections, geometry occlusion, fog volumes

Figure 4. *Alien indoor environment*

Ice Material layer, subsurface scattering

Figure 5. *Ice environment*

Achieving all three environments is a challenge as it's hard to optimize for levels with completely different characteristics.

- **Cinematographic quality rendering without hitting the Uncanny Valley**
  The closer you get to movies quality, the less forgiving the audience will be.

- **Dynamic light and shadows**
  Pre-computing lighting is crucial to many algorithms that improve performance and quality. Having dynamic light and shadows prevents us from using most of those algorithms because they often rely on static properties.

- **Support for multiple GPU and multiple CPU (MGPU & MCPU)**
  Development with multithreading and multiple graphic cards is much more complex and often it's hard to not scarify other configurations.

- **Game design requested a 21km×21km game play area**
  We considered doing this; but production, streaming, world persistence would not be worth the effort. We ended up having multiple levels with up to 4km×4km.

- **Target GPU from shader model 2.0 to 4.0 (DirectX10)**
  Starting with Shader Model 2.0 was quite convenient but DirectX10® development with early hardware and early drivers often slowed us down.

- **High Dynamic Range**
  We had good results with HDR in *Far Cry*, and for the realistic look we wanted to develop the game without the LDR limitations.

- **Dynamic environment (breakable)**
  This turned out to be one of the coolest features but it wasn't easy to achieve.

- **Developing game and engine together**
  That forced us to have the code always in some usable state. That's simple for a small project but becomes a challenge when doing on a large scale.

Our concept artists created many concept images in order to define the game's initial look but in order to ultimately define the feel of the game we produced a video. The external company Blur[3] studio produced with our input a few concept videos for us and that helped to come to consent on the look and feel we wanted to achieve.

Figure 6. *A frame from one of the concept videos from Blur (rendered off-line) for Crysis.*

8.3 Overview

In the remainder of this chapter we will first discuss the shader framework used by the new *CryEngine 2*. This area turned out to be a significant challenge for our large scale production. Then we will describe our solutions for direct and indirect lighting (including some of our design decisions). We can use specialized algorithms by isolating particular

[3] http://www.blur.com

lighting approach into a contained problem and solving it in the most efficient way. In that context, we approach direct lighting primarily from the point of view of shadowing (since shading can be done quite easily with shaders of varied sophistication). Indirect lighting can be approximated by ambient occlusion, a simple darkening of the ambient shading contribution. Finally we cover various algorithms that solve the level of detail problem. Of course this chapter will only cover but a few rendering aspects of our engine and many topics will be left uncovered – but it should give a good "taste" of the complexity of our system and allow us to dig in into a few select areas in sufficient details.

8.4 Shaders and Shading

8.4.1 Historical Perspective on CryEngine 1

In *Far Cry* we supported graphics hardware down to NVIDIA GeForce 2 which means we not only had pixel and vertex shader but also fixed function transform and lighting (T&L) and register combiner (pre pixel shader solution to blend textures) support. Because of that and to support complex materials for DirectX and OpenGL our shader scripts had complex syntax.

After *Far Cry* we wanted to improve that and refactored the system. We removed fixed function support and made the syntax more FX-like as described in [Microsoft07].

Very late in the project our renderer programmer introduced a new render path that was based on some über-shader approach. That was basically one pixel shader and vertex shader written in CG/HLSL with a lot of #ifdef. That turned out to be much simpler and faster for development as we completely avoided the hand optimization step. The early shader compilers were not always able to create shaders as optimal as humans could do but it was a good solution for shader model 2.0 graphics cards.

The über-shader had so many variations that compiling all of them was simply not possible. We accepted a noticeable stall due to compilation during development (when shader compilation was necessary) but we wanted to ship the game with a shader cache that had all shaders precompiled. We ended up playing the game on NVIDIA and on ATI till the cache wasn't getting new entries. We shipped *Far Cry* with that but clearly that wasn't a good solution and we had to improve that. We describe a lot more details about our first engine in [Wenzel05].

8.4.2 CryEngine 2

We decided to reduce a number of requirements for cleaner engine. As a result we removed support for OpenGL and fixed function pipeline support. This allowed us to make the shader scripts more FX format compatible. Then developing shaders became much more convenient and simple to learn.

We still had the problem with too many shader combinations and wanted to solve that. We changed the system by creating a shader cache request list. That list was gathered from all computers in the company over the network and it was used during the nightly shader cache compilation. However compilation time was long so we constantly had to reduce the amount of combinations.

We had the following options:

- Dynamic branching
- Reducing combinations and accepting less functionality
- Reducing combinations and accepting less performance
- Separating into multiple passes

We did that in multiple iterations and together with a distributed shader compilation we managed to compile all shaders for a build in about an hour.

8.4.3 3Dc™ for Normal Maps

The 3Dc™ texture format introduced by ATI [ATI04] allows compressing normal maps in one byte per texel with good quality and only little extra shader cost (reconstructing the z component). Uncompressed normal maps cost 4 bytes per texel (XYZ stored in RGB, one byte usually wasted for padding). In our new engine we decided to not do texture compression at load time. Textures become processed by our resource compiler tool and there we generate the mip levels and apply the compression. This way we get smaller builds and faster loading. For hardware that doesn't allow 3Dc™ compression we convert the 3Dc™ to DXT5 at load time. The formats are quite similar and conversion is simple. The minor quality loss is acceptable for low spec. Older NVIDIA cards have 3Dc™ emulation in the drivers so we don't have to take care of that (appears without visible quality loss, however, with this solution requires 2 byte per texel storage).

8.4.4 Per-Pixel Scene Depth

Using an early z pass can reduce per pixel shading cost because many pixel can be rejected based on the z value before a pixel shader needs to be executed. From beginning on we based our rendering on early z because we expected heavy pixel shader usage. For that we have to accept a higher draw call count. For many effects the depth value would be useful. As it wasn't possible to bind the z buffer we decided to output that value to a texture. At first we used the R16G16 texture format as this was available on all hardware and the 16 bit float quality was sufficient. Initially we had some use for the second channel but later we optimized that away. On ATI R16 was an option and to save some memory and bandwidth we used that format. We realized on some hardware the R16G16 is actually slower than the R32 format so we used R32 when R16 was not available. An even better option is using the z buffer directly as we don't need

extra memory and the early z pass can run faster (double speed without color write on some hardware). So we ended up using R16, R32 or even native z buffer – depending on what is available.

The depth value allows some tricks known from deferred shading. With one MAD operation and a 3 component interpolator it's possible to reconstruct the world space position. However for floating point precision it's better to reconstruct positions relative to the camera position or some point near to it. That is especially important when using 24bit or 16bit float in the pixel shader. By offsetting all objects and lights it's possible move the 0, 0, 0 origin near the viewer. Without doing this decals and animations can flicker and jump. We used the scene depth for per pixel atmospheric effects like the global fog, fog volumes and soft z buffered particles.

Shadow mask generation uses scene depth to reduce draw call count. For the water we use the depth value to soft clip the water and fade in a procedural shore effect. Several post processing effects like motion blur, Depth of Field and Edge blurring (EdgeAA) make use of the per pixel depth as well. We describe these effects in detail in [Wenzel07].

8.4.5 World Space Shading

In *Far Cry* we transformed the view and the light positions into tangent space (relative to the surface orientation). All data in the pixel shader was in tangent space so shading computations were done in that space. With multiple lights we were running into problems passing the light parameters over the limited amount of interpolators. To overcome this problem we switched to use world-space shading for all computations in *Crisis* (in actuality we use world-space shading with an offset in order to reduce floating point precision issues). The method was already needed for cube map reflections so code became more unified and shading quality improved as this space is not distorted as tangent space can be.

Parameters like light position now can be passed in pixel shader constants and don't need to be updated for each object. However when using only one light and simple shading the extra pixel cost is higher.

8.5 Shadows and Ambient Occlusion

8.5.1 Shadowing Approach in CryEngine 1

In our first title *Far Cry* we had shadow maps and projected shadows per object for the sun shadows. We suffered from typical shadow map aliasing quality issues but it was a good choice at that time. For performance reasons we pre-computed vegetation shadows but memory restrictions limited us to very blurry textures. For high end

hardware configurations we added shadow maps even to vegetation but combining them with the pre-computed solution was flawed.

We used stencil shadows for point lights as that were an easier and more efficient solution. CPU skinning allowed shadow silhouette extraction on the CPU and the GPU rendered the stencil shadows. It became obvious that this technique would become a problem the more detailed objects we wanted to render. It relied on CPU skinning, required extra CPU computation, an upload to GPU, extra memory for the edge data structures and had hardly predictable performance characteristics. The missing support for alpha-blended or tested shadow casters made this technique not even usable for the palm trees – an asset that was crucial for the tropical island look (Figure 7).

Figure 7. Far Cry *screenshot: note how the soft precomputed shadows combine with the real-time shadows*

For some time during development we had hoped the stencil shadows could be used for all indoor shadows. However the hard stencil shadows look and performance issues with many lights made us search for other solutions as well.

One of such solutions is to rely on light maps for shadowing. Light maps have the same performance no matter how many lights and allow a soft penumbra. Unfortunately what is usually stored is the result of the shading, a simple RGB color. That doesn't allow normal mapping. We managed to solve this problem and named our solution *Dot3Lightmaps* [Mittring04]. In this approach the light map stores an average light direction in tangent space together with an average light color and a blend value to lerp between pure ambient and pure directional lighting. That allowed us to render the diffuse contribution of static lights with soft shadows quite efficiently. However it was hard to combine with real-time shadows. After *Far Cry* we experimented with a simple modification that we named *Occlusion maps*. The main concept is to store the shadow mask value, a scalar value from 0 to 1 that represents the percentage of geometry occlusion for a texel. We stored the shadow mask of multiple lights in the light map texture and the usual four texture channels allowed four lights per texel. This way we rendered diffuse and specular contributions of static lights with high quality soft shadows while the light color and strength remained adjustable. We kept lights separate so combining with other shadow types was possible.

8.5.2 The Plan for CryEngine 2

The time seemed right for a clean unified shadow system. Because of the problems mentioned we decided to drop stencil shadows. Shadow maps offer high quality soft shadows and can be adjusted for better performance or quality so that was our choice. However that only covers the direct lighting and without the indirect lighting component the image would not get the cinematographic realistic look we wanted to achieve. The plan was to have a specialized solution for the direct and another one for the indirect lighting component.

8.5.3 Direct Lighting

For direct lighting we decided to apply shadow maps (storing depth of objects seen from the light in a 2D texture) only and drop all stencil shadow code.

8.5.3.1 Dynamic Occlusion Maps

To efficiently handle static lighting situations we wanted to do something new. By using some kind of unique unwrapping of the indoor geometry the shadow map lookup results could be stored into an occlusion map and dynamically updated. The dynamic occlusion map idea was good and it worked but shadows often showed aliasing as now we not only had shadow map aliasing but also unwrapping aliasing. Stretched textures introduced more artifacts and it was hard to get rid of all the seams. Additionally we still required shadow maps for dynamic objects so we decided to get the maximum out of normal shadow maps and dropped the caching in occlusions maps.

8.5.3.2 Shadow Maps with Screen-Space Randomized Look-up

Plain shadow mapping suffers from aliasing and has hard jagged edges (see first image in Figure). The PCF extension (percentage closer filtering) limits the problem (second image in Figure) but it requires many samples. Additionally at the time hardware support was only available on NVIDIA graphics cards such as GeForce 6 and 7 generation and emulation was even slower. We could implement the same approach on newer ATI graphics cards by using Fetch4 functionality (as described in [Isidoro06]).

Instead of adding more samples to the PCF filter we had the idea to randomize the lookup per pixel so that less samples result in similar quality accepting a bit of image noise. Noise (or grain) is part of any film image and the sample count offers an ideal property to adjust between quality and performance. The idea was inspired by soft shadow algorithms for ray tracing and already applied to shadow maps on GPU (See [Uralsky05] and [Isidoro06] for many details with regards to shadow map quality improvement and optimization).

The randomized offsets that form a disk shape can be applied in 2D when doing the texture lookup. When using big offsets the quality for flat surfaces can be improved by orienting the disk shape to the surface. Using a 3D shape like a sphere can have higher shading cost but it might soften bias problems.

To get acceptable results without too much noise multiple samples are needed. The sample count and the randomization algorithm can be chosen depending on quality and performance needs. We tried two main approaches: randomly rotated static kernel [Isidoro06] and another technique that allowed a simpler pixel shader.

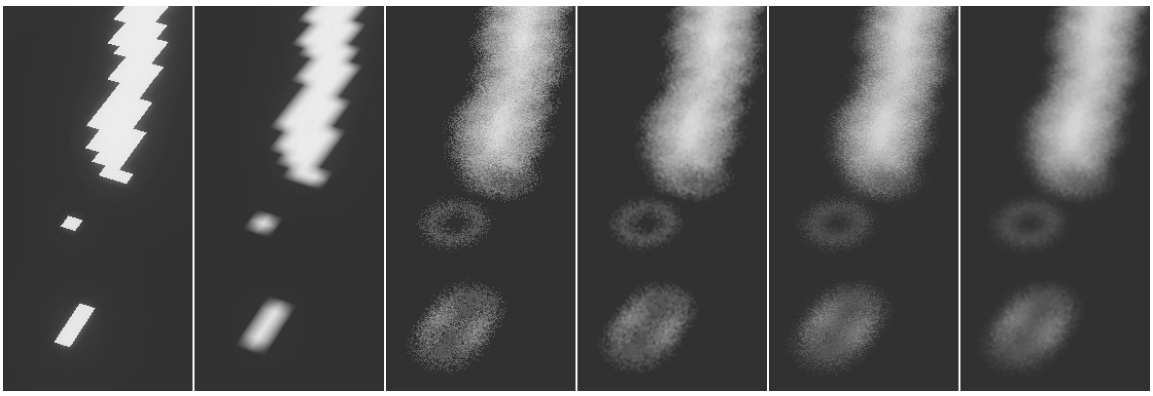


Figure 8. *Example of shadow mapping with varied resulting quality: from left to right: no PCF, PCF, 8 samples, 8 samples+blur, PCF+8 samples, PCF+8 samples+blur*

The first technique requires a static table of random 2D points and a texture with random rotation matrices. Luckily the rotation matrixes are small (2×2) and can be efficiently stored in a 4 component texture. As the matrices are orthogonal further compression is possible but not required. Negative numbers can be represented by the usual "scale and bias" trick (multiply the value by 2 and subtract 1) or by using floating point textures. We tried different sample tables and in the Figure 8 you can see an example of applying this approach to a soft disc that works quite well. For a disc shaped caster you would expect a filled disk but we haven't added the inner samples as the random rotation of those are less useful for sampling. The effect is rarely visible but to get more correct results we still consider changing it.

The simpler technique finds its sample positions by transforming one or two random positive 2D positions from the texture with simple transformations. The first point can be placed in the middle *(mx, my)* and four other points can be placed around using the random value *(x, y).*

(mx, my)
(mx+x, my+y)
(mx-y, my+x)
(mx-x, my-y)
(mx+y, my-x)

More points can be constructed accordingly but we found it only useful for materials rendered on low end hardware configurations (where we would want to keep the sample count low for performance reasons).

Both techniques also allow adjusting the kernel size to simulate soft shadows. To get proper results this kernel adjustment would be dependent on the caster distance and the light radius but often this can be approximated much easier. Initially we randomized by using a 64x64 texture tiled with a 1:1 pixel mapping over the screen (Figure 9)

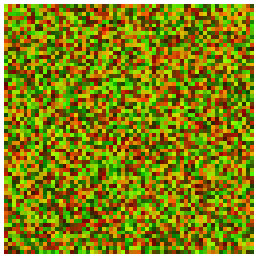


Figure 9*. An example of the randomized kernel adjustment texture*

This texture (Figure 9) was carefully crafted to appear random without recognizable features and with most details in the higher frequencies. Creating a random texture is fairly straight-forward; we can manually reject textures with recognizable features and we can maximize higher frequencies applying a simple algorithm that finds a good pair of neighbor pixels that can be swapped. A good swapping pair will increase high frequencies (computed by summing up the differences). While there are certainly better methods to create a random texture with high frequencies), we only describe but this simple technique as it served our purposes.

Film grain effect is not a static effect so we could potentially animate the noise and expect it to hide low sample count even more. Unfortunately the result was perceived as a new type of artifact with low or varying frame rate. Noise without animation looked pleasing for static scenes; however with a moving camera some recognizable static features in the random noise remained on the screen.

8.5.3.3 Shadow Maps with Light-Space Randomized Look-up

Fortunately we found a good solution for that problem. Instead of projecting the noise to the screen we projected a mip-mapped noise texture in world space in the light/sun direction. In medium and far distance the result was the same but because of bilinear magnification the nearby shadow edges became distorted and no longer noisy. That looked significantly better – particularly for foliage and vegetation, where the exact shadow shape was hard to determine.

8.5.3.4 Shadow Mask Texture

We separated the shadow lookup from shading in our shaders in order to avoid the instruction count limitations of Shader Model 2.0, as well as to reduce the number of resulting shader combinations and be able to combine multiple shadows. We stored the 8 bit result of the shadow map lookup in a screen-space texture we named *shadow*

mask. The 4 channel 32 bit texture format offers the required bit count and it can be used as a render target. As we have 4 channels we can combine up to 4 light contributions in a texel.

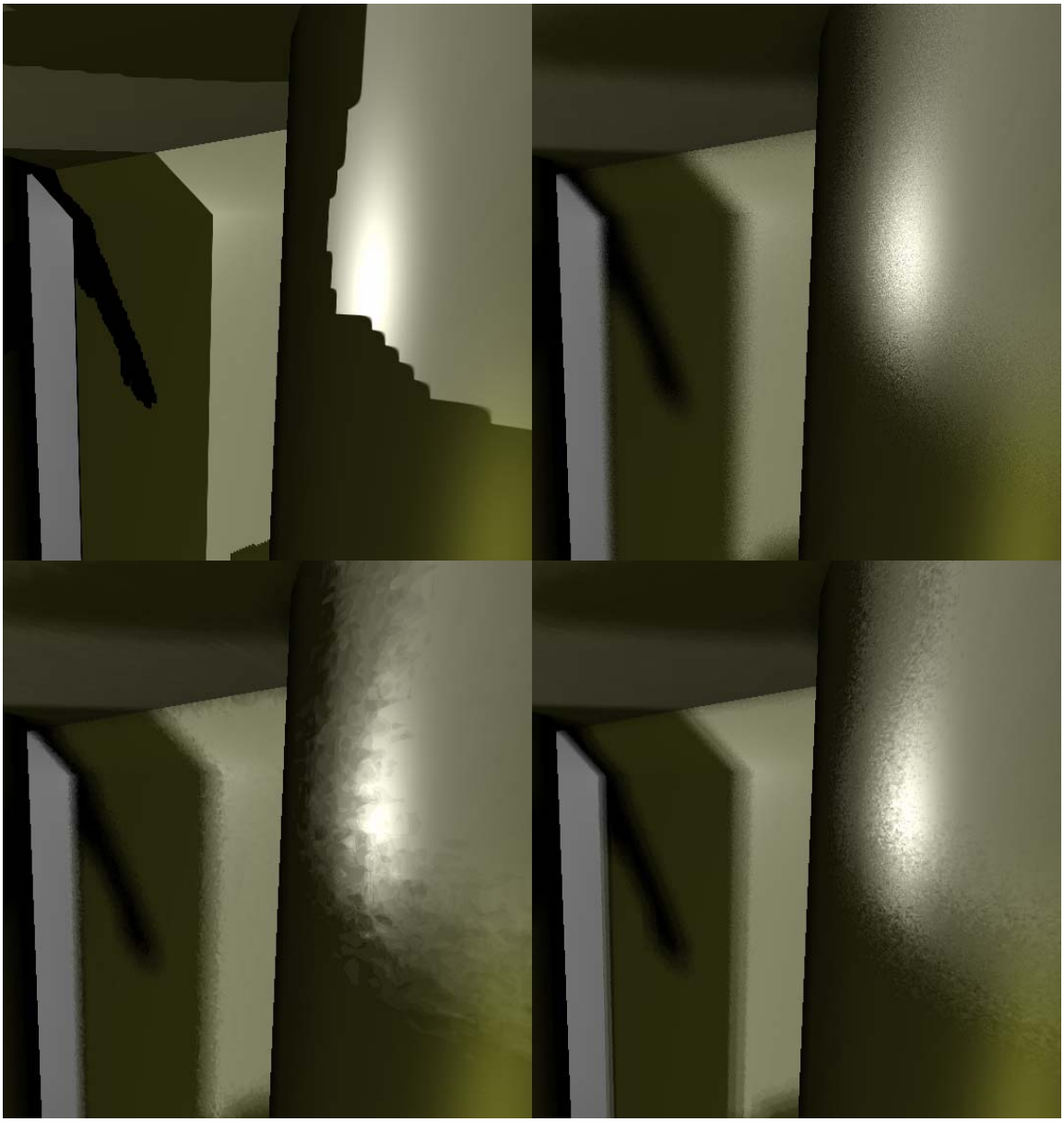


Figure 10. *Example of shadow maps with randomized look-up. Left top row image: no jittering 1 sample, right top row image: screen space noise 8 samples, left bottom: world space noise 8 samples, right bottom: world space noise with tweaked settings 8 samples*

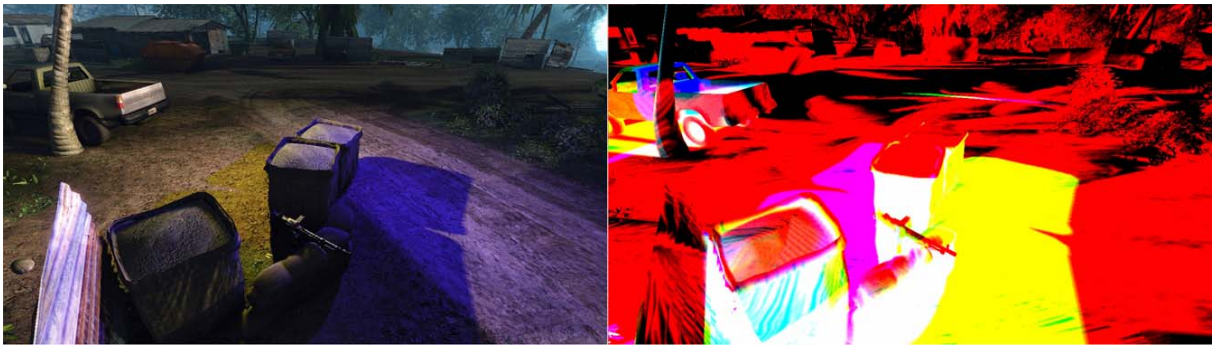


Figure 11. *Example of the shadow mask texture for a given scene: Left: final rendering with sun (as a shadow caster) and two shadow-casting lights, right: light mask texture with three lights in the RGB channels*

Figure 12. *Example of the shadow mask texture for a given scene - Red, Green and Blue channel store the shadow mask for 3 individual lights*

In the shading pass we bind this texture and render multiple lights and the ambient at once. We could have used the alpha channel of the frame buffer but then we would have more passes and draw call count would raise a lot. For opaque objects and alpha test surfaces the shadow mask is a good solution but it doesn't work very well for alpha blended geometry. All opaque geometry is represented in the depth buffer but alpha blended geometry is not modifying the depth buffer. Transparent geometry requires normal shadow map lookup in the shader.

8.5.3.5 Shadow Maps for Directional Light Sources

In *Far Cry* we had only a few shadow casting objects and each had its own shadow map. For many objects its better to combine them on one shadow map. A simple parallel projection in the direction of the light works but near the viewer the shadow map resolution is quite low and then shadows appear blocky. Changing the parameterization like finding a projection matrix that moved more resolution near the viewer is possible but not without problems. We tried trapezoidal shadow maps ([MT04]) (TSM) and perspective shadow maps ([SD02]) (PSM).

We had more success with cascaded shadow maps (CSM) where multiple shadow maps of the same resolution cover the viewer area with multiple projections. Each projection is enclosed by the previous one with decreasing world to texel ratio. That technique was giving satisfactory results but wasted some texture space. That was because the projection only roughly concentrated to the area in front of the viewer. To find proper projection the view frustum (reduced by the shadow receiving distance) can

be sliced up. Each shadow map needs to covers one slice. Slices farther away can cover bigger world space areas. If the shadow map projection covers the slices tightly then minimal shadow map area is wasted.

With earlier shadow techniques we already had aliasing of the shadow maps when doing camera movements and rotations. For PSM and TSM we haven't been able to solve the issue but for CSM and its modification it was possible. We simply snapped the projections per shadow map texel and that resulted in a much cleaner look.

8.5.3.6 Deferred Shadow Mask Generation

The initial shadow mask generation pass required rendering of all receiving objects and that resulted in many draw calls. We decoupled shadow mask generation from the receiver object count by using deferred techniques. We basically render a full screen pass that binds the depth texture we created in the early z pass. Simple pixel shader computations give us the shadow map lookup position based on the depth value. The indirection over the world-space position is not needed.

As mentioned before we used multiple shadow maps so the shadow mask generation pixel shader had to identify for each pixel in which shadow map it falls and index into the right texture. Indexing into a texture can be done with DirectX10 texture arrays feature or by offsetting the lookup within a combined texture.

By using the stencil buffer we were able to separate processing of the individual slices and that simplified the pixel shader. Indexing was not needed any more. The modified technique runs faster as less complex pixel shader computations need to be done. It also carves away far distant areas that don't receive shadows.

8.5.3.7 Unwrapped Shadow Maps for Point Lights

The usual shadow map approach for point light sources require a cube map texture lookup. But then hardware PCF cannot be used and on cube maps there is much less control for managing the texture memory.

We unwrapped the cube map into six shadow maps by separating the six cases with the stencil buffer, similar we did for CSM. This way we transformed the point light source problem to the projector light problem. That unified the code and resulted in less code to maintain and optimize and less shader combinations.

8.5.3.8 Variance Shadow Maps

For terrain we initially wanted to pre-compute a texture with start and end angle. We also tried to update an occlusion map in real-time with incremental updates. However the

problem has always been objects on the terrain. Big objects, partly on different terrain sectors required proper shadows. We tried to use our normal shadow map approach and it gave us a consistent look that wasn't soft enough. Simply making the randomized lookup with a bigger radius would be far too noisy. Here we tried variance shadow maps [DL06] and this approach has worked out nicely. The usual drawback of variance shadow maps arises with multiple shadow casters behind each other but that's a rare case with terrain shadows.

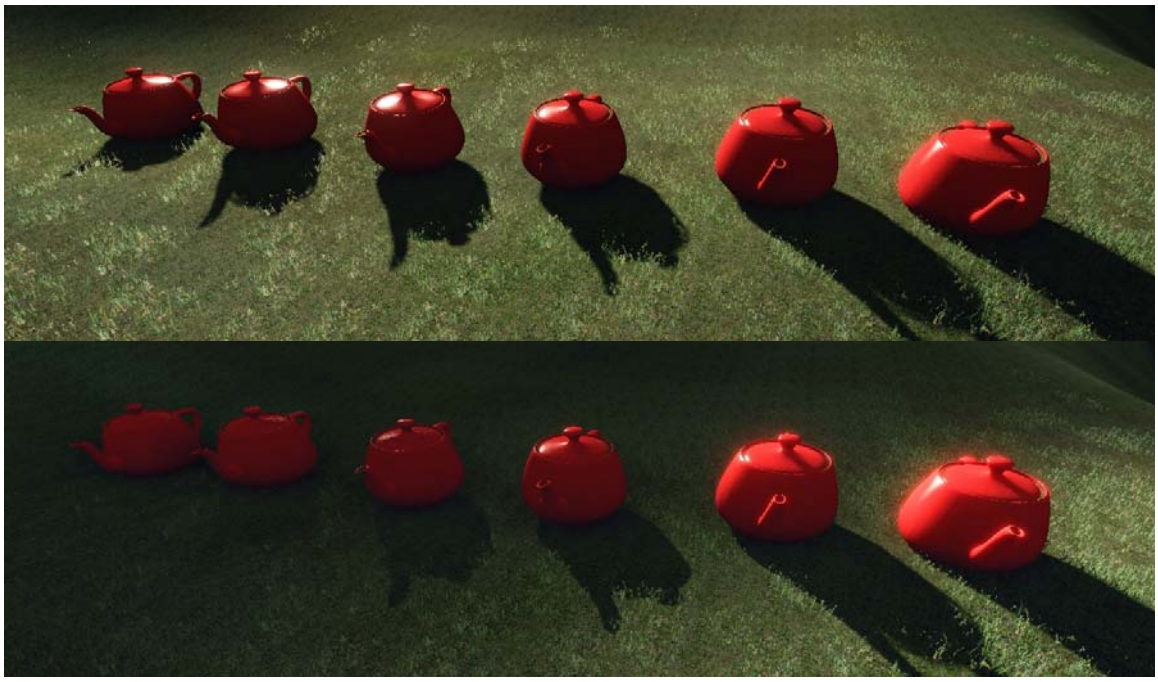


Figure 13. *Example of applying variance shadow maps to a scene. Top image: variance shadow maps aren't used (note the hard normal shadows), bottom image: with variance shadow maps (note how the two shadow types combine)*

8.5.4 Indirect Lighting

The indirect lighting solution can be split in two sub-problems: the processing intensive part of computing the indirect lighting and the reconstruction of the data in the pixel shader (to support per-pixel lighting).

8.5.4.1 3D Transport Sampler

For the first part we had planned to develop a tool called *3D transport sampler*. This tool would make it possible to compute the global illumination data distributed on multiple machines (for performance reasons). Photon mapping ([Jensen01]) is one of the most accepted methods for global illumination computation. We decided to use this method because it can be easily integrated and delivers good results quickly. The photon

Figure 14. *Real-time ambient maps with one light source*

mapper was first used to create a simple light map. The unwrapping technique in our old light mapper was simple and only combined triangles that were connected and had a similar plane equation. That resulted in many small 2D blocks we packed into multiple textures. When used for detailed models it became inefficient in texture usage and it resulted in many small discontinuities on the unwrapping borders. We changed the unwrapping technique so it uses the models UV unwrapping as a base and modifies the unwrapping only where needed. This way the artist had more control over the process and the technique is more suitable for detailed models. We considered storing Dot3Lightmaps (explained earlier) but what we tried was a method that should result in better quality. The idea was to store light contributions for four directions oriented to the surface. This is similar to the technique that was used in Half-Life 2 ([McTaggart04]) but there only three directions were used. The more data would allow better quality shading. The data would allow high quality per-pixel lighting and accepting some approximations it could be combined with real-time shadows. However storage cost was huge and computation time was high so we aborted this approach. Actually our original plan was to store some light map coefficients per texel and others per vertex. Together with a graph data structure that is connected to the vertices it should be possible to get dynamic indirect lighting. Low frequency components of the indirect lighting could be stored in the vertices and high frequency components like sharp corners could be stored per texel. Development time was critical so this idea was dropped.

8.5.4.2 Real-Time Ambient Map (RAM)

As an alternative we chose a much simpler solution which only required storing one scalar ambient occlusion value per texel. Ambient occlusion ([ZIK98, Landis02]) can be computed by shooting rays in all directions – something that was reusable from the photon mapper. The reconstruction in the shader was using what was available: the

texel with the occlusion value, the light position relative to the surface, the light color and the surface normal. The result was a crude approximation of indirect lighting but the human eye is very forgiving for indirect lighting so it worked out very well.

To support normal maps some average light direction is needed and because of the lack of something better the light direction blended with the surface normal was used. This way the normal maps still can be seen and shading appear to have some light angle dependency. Having ambient brightness, color and attenuation curve adjustable allowed designers to tweak the final look.

The technique was greater extended to take portals into account, to combine multiple lights and to support the sun. For huge outdoor areas computing the RAM data for every surface wouldn't be feasible so we approached that differently.

8.5.4.3 Screen-Space Ambient Occlusion

One of our creative programmers had the idea to use the z buffer data we already had in a texture to compute some kind of ambient occlusion. The idea was tempting because all opaque objects could be handled without special cases in constant time and constant memory. We also could remove a lot of complexity in many areas. Our existing solutions worked but it we had issues to handle all kind of dynamic situations.

The approach was based on sampling the surrounding of a pixel and with some simple depth comparisons it was possible to compute a darkening factor to get silhouettes around objects. To get the ambient occlusion look this effect was limited to only nearby receivers. After several iterations and optimizations we finally had an unexpected new feature and we called it "Screen-Space Ambient Occlusion" (SSAO).

We compute the screen-space ambient occlusion in a full screen pass. We experimented by applying it on ambient, diffuse and specular shading but we found it works best on ambient only. That was mostly because it changed the look away from being realistic and that was one of our goals.

To reduce the sample count we vary the sample position for nearby pixels. The initial sample positions are distributed around the origin in a sphere and the variation is achieved by reflecting the sample positions on a random 3D plane through the origin.

n: the normalized random per pixel vector from the texture
i: one of the 3D sample positions in a sphere

```
float3 reflect( float3 i,  float3 n ) { return i - 2 * dot(i, n) * n;
```

The reflection is simple to compute and it's enough to store the normalized plane normal in a texture.

Figure 15. *Screen-Space Ambient Occlusion in a complete ambient lighting situation (note how occluded areas darken at any distance)*

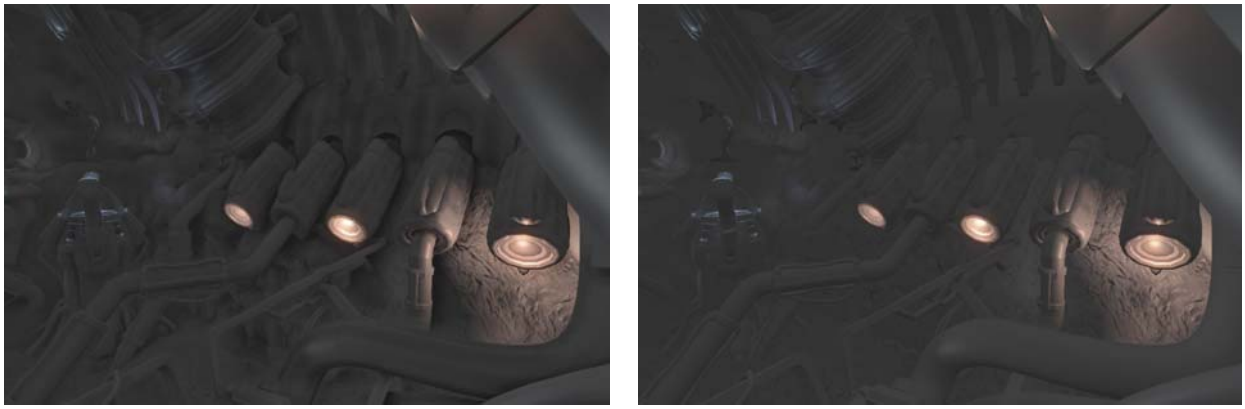


Figure 16. *Sample scene A with special material setup to visualize SSAO (left: with SSAO, right: without SSAO)*

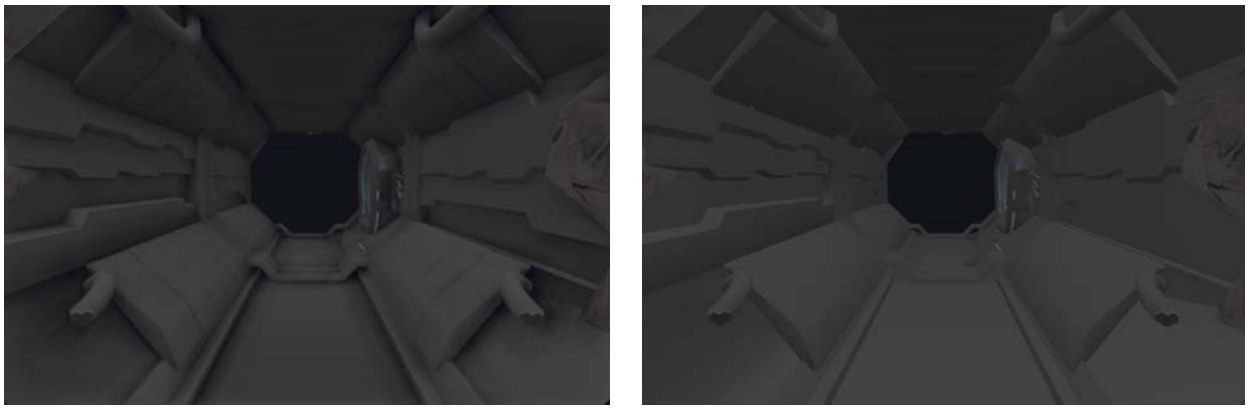


Figure 17. *Sample scene B with special material setup to visualize SSAO (left: with SSAO, right: without SSAO)*

Figure 18. *Sample scene A with special material setup to visualize SSAO (left: with SSAO, right: without SSAO)*

8.6 Level of Detail

8.6.1 Situation

Level of Detail (LOD) is especially important if the rendering complexity cannot be easily restricted. Most games have either quite limited view range often realized with fog or strong occlusion set up by level designers. That's why many games are dominated by indoor environments but in *Far Cry* we wanted to show big landscapes with many details without restricting the players view or position. In *Crysis* we kept the view range but added more objects with more variety and higher quality. Quality means more complex pixel and vertex shaders, higher resolution textures, new texture types (e.g. subsurface) and more vertices. As we additionally decided to use an early z pass and real-time shadows we have to pay an even higher price for each object. Because of the higher draw call count this is mainly a CPU burden and that is one of the areas where DirectX10 is better. In *Far Cry* we simply switched between artist-created LOD models. We also used impostors for vegetation and improved them for *Crysis* but that's beyond the scope this chapter.

In *Crysis* we considered using a smooth LOD transition based on moving vertices. Such techniques often introduce many restrictions for the asset creation. Without such restrictions assets often can be more optimal and created more quickly. That is especially true for vegetation rendered alpha-tested or alpha-blended. For some time we had no transition between LODs and for demo purposes we disabled lower LOD levels completely. Demo machines have high spec hardware and there the smaller LOD levels had no increased effect. Lower LODs are usually small on the screen so per pixel cost is low. Having many LOD levels can be even counterproductive, as those cannot be instanced together and, so we suffer from higher draw call count.

8.6.2 Dissolve

One of our programmers finally had the idea of a soft LOD transition based on dissolving the object in the early z pass. As we later on render with z equal comparison we only had to adjust the early z pass. That was not completely true as surfaces can have exactly the same z value and then with additive blending those pixels would become twice as bright. However as the first rendering pass of each object has frame buffer blending disabled the problem should only occur with subsequent passes. As we can combine multiple lights in one pass this is a rare case anyway.

The dissolve texture is projected in screen space, and by combining the random value from the texture with a per object transition value, the pixels are rejected with the texkill operation or simple alpha-test. With the Alpha2Coverage feature and full scene anti-aliasing (FSAA) of modern cards that can be even done on a sub-pixel level. Even without FSAA the dissolve is not that noticeable if we enable our edge-blurring post processing effect.

Initially we had the transition state only depend on object distance but objects that are in transition are slower to render and for quality reasons it's better to hide it. That's why we added code to finish started transitions within a defined small amount of time. We not only use the dissolve for transitions between 3D objects but also to fade out far away objects and to hide the transition to impostors.

8.6.3 Water Surface LOD

The ocean or big water surfaces in general have some unique properties that can be used by specialized render algorithms. Our initial implementation that we used in *Far Cry* was based on a simple disk mesh that moved around with the player. Pixel shading, reflections and transparency defined the look. However we wanted to have real 3D waves, not based on physical simulation but a cheap procedural solution. We experimented with some FFT based ocean waves simulation ([Jensen01a], [Tessendorf04]).

To get 3D waves vertex position manipulation was required and the mesh we used so far wasn't serving that purpose.

8.6.4 Square Water Sectors

The FFT mentioned earlier only outputs a small sector of the ocean and to get a surface till the horizon we rendered the mesh multiple times. Different LODs had different index buffers but they all referenced to one vertex buffer that had the FFT data. We shared the vertex buffer to save performance but for better quality down sampling would be needed. To reduce aliasing artifacts in the distance and to limit the low polygonal look in the near we faded out the perturbation for distant vertices and limited the perturbation in the near. The algorithm worked but many artifacts made us search for a better solution.

8.6.5 Screen-Space Tessellation

We tried a brute force approach that was surprisingly simple and worked out very well. We used a precomputed screen space tessellated quad and projected the whole mesh onto the water surface. This ensures correct z buffer behavior and even clips away pixels above the horizon line. To modify the vertex positions with the FFT wave simulation data we require vertex texture lookup so this feature cannot be used on all hardware.

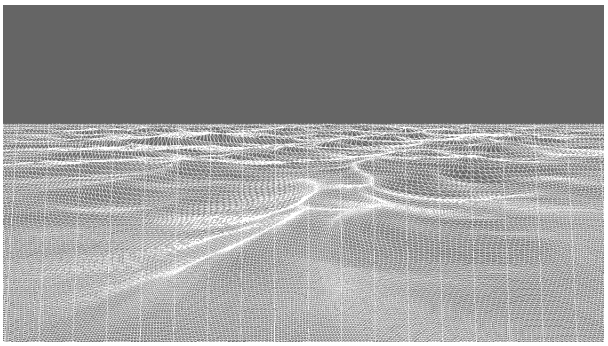


Figure 2. *Screen-space tessellation in wireframe*

The visible vertical lines in the wireframe are due to the mesh stripification we do for better vertex cache performance. The results looked quite promising however vertices on the screen border often moved farther away from the border and that was unacceptable. Adding more vertices even outside of the screen would solve the problem but attenuating the perturbations on the screen border are hardly noticeable and have only minimal extra cost.

Figure 3. *Left: screen space tessellation without edge attenuation (note the area on the left not covered by water), right: screen space tessellation with edge attenuation*

For better performance we reduced the mesh tessellation. Artifacts remained acceptable even with far less vertices. Tilting the camera made it a slightly worse but not as much as we expected. The edge attenuation made the water surface camera dependent and that was bad for proper physics interaction. We had to reduce the wave amplitude a lot to limit the problem.

8.6.6 Camera Aligned

The remaining issues aliasing artifacts and physics interaction bothered our shader programmer and he spent some extra hours finding a solution for this. This new method used a static mesh like the one before. The mesh projection changed from a perspective to a simple top down projection. The mesh is dragged around with the camera and the offset is adjusted to get most of the mesh in front of the camera. To render up to the horizon line the mesh borders are expanded significantly. Tessellation in that area is not crucial as perturbation can be faded to 0 before that distance.

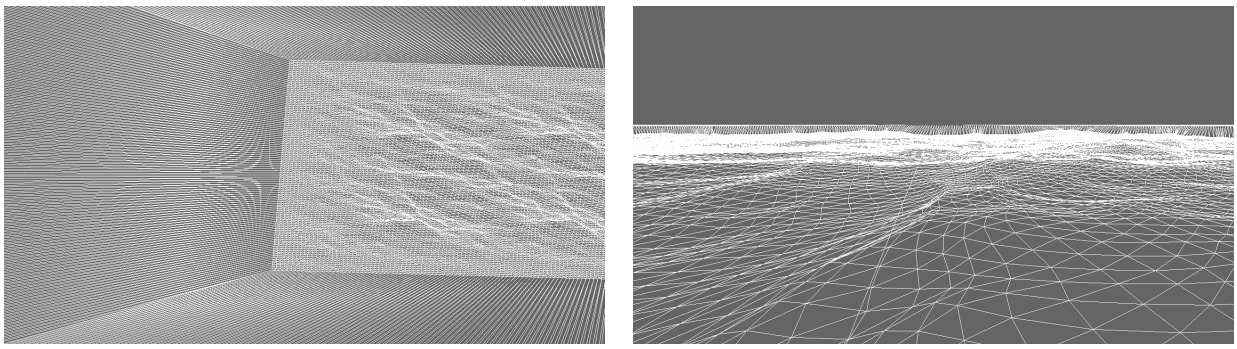


Figure 204. *Camera aligned water mesh in wire frame. Left: camera aligned from top down, right: camera aligned from viewer perspective*

The results of this method are superior to the screen space ones which becomes mostly visible in motion with subtle camera movement. Apart from the distance attenuation the wave extent is now viewer independent and as the FFT data is CPU accessible physics interactions are now possible.

Figure 21. *Left: Camera aligned, right: screen space tessellation as comparison*

8.6 Conclusion

Through some intricate path we not only found our next generation engine but we also learned a lot. That learning process was necessary to find, validate and compare different solutions so in retro perspective it can be classified to research. Why we chose certain solutions in favor of others is mostly because of quality, production time, performance and scalability. *Crysis*, our current game, is a big scale production and to handle this the production time is very important. Performance of a solution is hardware dependent (e.g. CPU, GPU, memory) so on a different platform we might have to reconsider. The current engine is streamlined for a fast DirectX9/DirectX10 card with one or multiple CPU cores.

Having the depth from the early z pass turned out to be very useful; many features now rely on this functionality. Regular deferred shading also stores more information per pixel like the diffuse color, normal and other material properties. For the alien indoor environment that would probably be the best solution but other environments would suffer from that decision. In a one light source situation deferred shading simply cannot play out its advantages.

8.6 Acknowledgements

This presentation is based on the passionate work of many programmers, artist and designers. Special thanks I would like to contribute to Vladimir Kajalin, Andrey Khonich, Tiago Sousa, Carsten Wenzel and Nick Kasyan. Because we have been launch partners with NVIDIA we had on-site help not only for G80 DirectX9 and DirectX10 issues. Special thanks to those NVIDIA engineers namely Miguel Sainz, Yury Uralsky and Philip Gerasimov. Leading companies of the industry Microsoft, AMD, Intel and NVIDIA and many others have been very supportive. Additional thanks to Natalya Tatarchuk and Tim Parlett that helped me to get this done.

8.7 References

[ATI04] ATI 2004, Radeon X800 3Dc™ Whitepaper
http://ati.de/products/radeonx800/3DcWhitePaper.pdf

[DL06] DONNELLY W. AND LAURITZEN A. 2006. Variance shadow maps. In Proceedings of the 2006 ACM SIGGRAPH Symposium on Interactive 3D graphics and games, pp. 161-165. Redwood City, CA

[ISIDORO06] ISIDORO J. 2006. Shadow Mapping: GPU-based Tips and Techniques. GDC presentation. http://ati.amd.com/developer/gdc/2006/Isidoro-ShadowMapping.pdf

[JENSEN01] JENSEN, H. W. 2001. Realistic image synthesis using photon mapping, A. K. Peters, Ltd., Natick, MA.

[JENSEN01a] JENSEN, L. 2001, Deep-Water Animation and Rendering, Gamasutra article
http://www.gamasutra.com/gdce/2001/jensen/jensen_pfv.htm

[LANDIS02] LANDIS, H., 2002. RenderMan in Production, ACM SIGGRAPH 2002 Course 16.

[MICROSOFT07] MICROSOFT DIRECTX SDK. April 2007.
http://www.microsoft.com/downloads/details.aspx?FamilyID=86cf7fa2-e953-475c-abde-f016e4f7b61a&DisplayLang=en

[MT04] MARTIN, T. AND TAN, T.-S. 2004. Anti-aliasing and continuity with trapezoidal shadow maps. In proceedings of Eurographics Symposium on Rendering 2004, pp. 153–160, 2004.

[MCTAGGART04] MCTAGGART, G. 2004. Half-Life 2 Shading, GDC Direct3D Tutorial
http://www2.ati.com/developer/gdc/D3DTutorial10_Half-Life2_Shading.pdf

[MITTRING04] MITTRING, M. 2004. Method and Computer Program Product for Lighting a Computer Graphics Image and a Computer. US Patent 2004/0155879 A1, August 12, 2004.

[SD02] STAMMINGER, M. AND DRETTAKIS, G. 2002. Perspective shadow maps. In SIGGRAPH 2002 Conference Proceedings, volume 21, 3, pages 557–562, July 2002

[TESSENDORF04] TESSENDORF, J. 2004. Simulating Ocean Surfaces. Part of ACM SIGGRAPH 2004 Course 32, The Elements of Nature: Interactive and Realistic Techniques, Los Angeles, CA

[URALSKY05] URALSKY, Y. 2005. Efficient Soft-Edged Shadows Using Pixel Shader Branching. In GPU Gems 2, M. Pharr, Ed., Addison-Wesley, pp. 269 – 282.

[WENZEL05] WENZEL C. 2005. Far Cry and DirectX. GDC presentation, San Francisco, CA
http://ati.amd.com/developer/gdc/D3DTutorial08_FarCryAndDX9.pdf

[WENZEL06] WENZEL, C. 2006. Real-time Atmospheric Effects in Games. Course 26: Advanced Real-Time Rendering in 3D Graphics and Games. Siggraph, Boston, MA. August 2006 http://ati.amd.com/developer/techreports/2006/SIGGRAPH2006/Course_26_SIGGRAPH_2006.pdf

[WENZEL07] WENZEL C. 2007. Real-time Atmospheric Effects in Games Revisited. Conference Session. GDC 2007. March 5-9, 2007, San Francisco, CA. http://ati.amd.com/developer/gdc/2007/D3DTutorial_Crytek.pdf

[ZIK98] ZHUKOV, S., IONES, A., AND KRONIN, G. 1998. An ambient light illumination model. In Rendering Techniques '98 (Proceedings of the Eurographics Workshop on Rendering), pp. 45–55.

Chapter 9

Real-Time Isosurface Extraction Using the GPU Programmable Geometry Pipeline

Natalya Tatarchuk [14] Jeremy Shopf[15] Christopher DeCoro[16]
Game Computing Applications Group
AMD GPG (O-CTO)
Princeton University

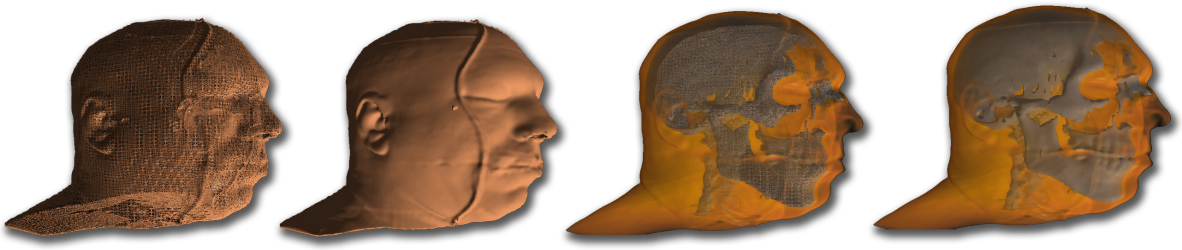


Figure 1. *We show the result of extracting a series of highly detailed isosurfaces at interactive rates. Our system implements a hybrid cubes-tetrahedra method, which leverages the strengths of each as applicable to the unique architecture of the GPU. The left pair of images (wireframe and shaded, using a base cube grid of 64^3) show only an extracted isosurface, while the right pair displays an alternate isosurface overlayed with a volume rendering.*

9.1 Introduction

In this chapter we present a set of approaches that allow interactive exploration of scalar in real-time. We implement our methods using massively parallel architectures available to average consumers - the latest commodity Graphics Processing Units (GPUs). Recent generations of consumer GPU architectures are designed for high efficiency and high computational load, along with effective use of coherency and extensive memory bandwidth requirements. Furthermore, with the advent of programmable GPU pipelines,

[14] natalya.tatarchuk@amd.com
[15] jeremy.shopf@amd.com
[16] cdecoro@cs.princeton.edu

we can take advantage of the massive parallelism of the GPUs, rather than relying on significantly slower scaling CPU multi-core architecture. In this work, we take advantage of huge memory bandwidth available on the most recent generation of consumer GPU to be able to process extensive datasets, such as the Virtual Human Project dataset (taking up more than 576MB of video memory) at extremely fast frame rates. We utilize efficient pixel and geometry processing taking advantage of dynamic load balancing of the latest GPU architectures (such as ATI Radeon HD 2000 and 3000 series), and build our system to fully utilize Shader Model 4.0 capabilities available with DirectX® 10 GPU pipeline.

Our system combines direct volume rendering via ray-casting with isosurface extraction directly on GPU. The volume rendering approach is reformulated to take advantage of parallel pixel processing of the GPU pipelines. The isosurface extraction sub-system takes advantage of the novel DirectX 10 GPU pipeline for dynamic surface extraction in real-time. This framework is able to process individual voxel portions of the input dataset in parallel using geometry shaders. Our approach will scale with the number of parallel SIMD and texture units in a GPU generation.

We have developed a technique for real-time volume data analysis by providing an interactive user interface for designing material properties for organs in the scanned volume. We provide an interactive user-driven material design system for quick and intuitive organ classification based on material properties of the dataset. Intelligently combining isosurface extraction with direct volume rendering in a single system allows for surface properties as well as for the context of tissues surrounding the region and gives better context for navigation. Our application can be used with CT and MRI scan data, or with variety of other medical applications.

The pipeline for our visualization system presented in this chapter is as follows:

- We start by collecting the scalar data. In our case, we simply used an existing data set from the Visible Human Project, collected by the National Institute of Medicine.
- We then proceed to preprocess the data to compute gradients which are used for rendering correct lighting information at run-time. This is done in an offline process.
- Once we have extracted and preprocessed the data, we can proceed to render it, using our on-demand isosurface extraction on GPU and interactive volume rendering via ray-casting
- At any point we can also interactively classify features by using our material user interface, designed to interactively specify, edit and save custom 2D transfer functions for each dataset.

9.2 Efficient Isosurface Extraction and Rendering on GPU

An implicit surface representation, as opposed to explicit representation with a polygon mesh or parametric surface, is frequently the most convenient form for many modeling tasks. The high computational expense of extracting explicit isosurfaces from implicit functions, however, has made such operations a frequent candidate for GPU acceleration. Now with recent advancements in GPU architectures, we will demonstrate how an intuitive extraction algorithm can be implemented efficiently using a hybrid

marching cubes/marching tetrahedra approach. We are able to leverage the strengths of each method as applied to the unique computational environment of the GPU, and in doing so, achieve real-time extraction of detailed isosurfaces from high-resolution volume datasets. We are also able to perform adaptive surface refinement directly on the GPU without lowering the parallelism of our algorithm. In addition, we show that the complementary technique of inverse quadratic interpolation significantly improves surface quality.

Implicit functions are a powerful mechanism for modeling and editing geometry. Rather than the geometry of a surface given explicitly by a triangle mesh, parametric surface, or other boundary representation, it is defined implicitly by an arbitrary continuous function $f(x), x \in \Re^3$. By defining an arbitrary constant c, (referred to as an isovalue, and frequently 0), we can define as our surface as the set of all points (isosurface) for which $f(x) = c$. For simplicity, we will frequently make the substitution $F(x) = f(x) - c$, without loss of generality.

Aside from the area of medical visualization, deformable isosurfaces, implemented with level-set methods, have demonstrated a great potential in visualization for applications such as segmentation, surface processing, and surface reconstruction. Specifically, these hold immense importance in the area of fluid dynamics and rendering applications ranging to simulating complex fluid flows around objects to detailed fluid dynamics for liquids interacting with objects. Isosurfaces are normally displayed using computer graphics, and are used as data visualization methods in computational fluid dynamics (CFD), allowing engineers to study features of a fluid flow (gas or liquid) around objects, such as aircraft wings. An isosurface may represent an individual shockwave in supersonic flight, or several isosurfaces may be generated showing a sequence of pressure values in the air flowing around a wing. Isosurfaces tend to be a popular form of visualization for volume datasets since they can be rendered by a simple polygonal model, which can be drawn on the screen very quickly. Numerous other disciplines that are interested in three dimensional data often use isosurfaces to obtain information about pharmacology, chemistry, geophysics and meteorology.

The advantage to the above representation is to allow arbitrary definition of the underlying function – and thus the implied surface – in a simple and direct manner. Such a representation can easily have arbitrary and dynamic topology, modified using Boolean and arithmetic operators, analyzed using traditional signal processing techniques, and used with simulations of natural phenomena. To perform these operations using traditional, explicit modeling would be impractically complex. The disadvantage, however, is that such representations pose a challenge to render isosurfaces directly at real-time rates – especially given the rasterization pipelines used in GPUs, which are designed for triangle input data. Several classes of rendering techniques exist, such as direct volume rendering (volumes and implicit functions are synonymous in this context), which renders the volume without an intermediate boundary representation. Among such techniques, Westerman and Ertl [1] demonstrated a method for texture-based rendering of volume datasets defined on a uniform regular grid; later work proposed a generalized method for rendering volumes defined over tetrahedral grids [2].

While such techniques are limited to rendering applications, an alternate class of methods extracts an intermediate explicit isosurface (usually a triangle mesh) which can

be used for further processing and analysis in addition to rendering. Examples include later use for collision detection, shadow casting, and animation. The most commonly used algorithms for isosurface extraction are derivatives of the marching cubes (MC) algorithm [3] and the closely related marching tetrahedra (MT) algorithm [4]. The algorithm we present is a hybrid of these two methods, such that we leverage the strengths of each method as applicable to the unique constraints and benefits of the GPU architecture.

Isosurface extraction is typically a compute-intensive method. Isosurface extraction on GPU has been a topic of extensive research for the last several years. Much work has been done to generate highly efficient renderings at high frame rates. Prior to the DirectX10 generation of GPUs, the programming model lacked support for programmable primitive processing and the ability to output geometric quantities for later re-use. Thus previous work lacked the ability to generate polygonal surface directly on the GPU and re-use it for subsequent computation, such as collision detection or optimization of volumetric rendering for surrounding organs. Much of the extraction work was redundantly performed regardless of whether the isovalue was dynamically changing or not, resulting in wasted computation. Nonetheless, a number of researchers succeeded at fast, interactive isosurface rendering.

Pascucci [5] rendered tetrahedra to the GPU, using the MT algorithm in the vertex shader to re-map vertices to surface positions. Subsequently, Klein et al. [6] demonstrated a similar tetrahedral method using pixel shaders, which at the time provided greater throughput. Other researchers instead implemented the marching cubes algorithm [7,8]. For an broad overview of both direct rendering and extraction methods, see the survey by Silva et al. [9].

All of these methods, however, were forced to use contrived programming techniques to escape the limitations of the previously restrictive GPU architecture. Using the geometry shader stage, executing on primitives post vertex shader and prior to rasterizer processing, we are able to generate and to cull geometry directly on the GPU. This provides a more natural implementation of marching methods. With our approach the isosurface is dynamically extracted directly on GPU and constructed in polygonal form and can be directly used post-extraction for collision detection or rendering and optimization. The resulting polygonal surface can also be analyzed for geometric properties, such as feature area, volume and size deviation, which is crucial for semi-automatic tumor analysis, for example, as used in colonoscopy. Using our pipeline provides a direct method to reuse the result of geometry processing, in the form of a *stream out* option, which stores output triangles in a GPU buffer after the geometry shader stage. This buffer may be reused arbitrarily in later rendering stages, or even read back to the host CPU.

In our work we present a hybrid method that leverages the strengths of both marching cubes and marching tetrahedra, relative to the unique abilities and constraints of the latest GPU architecture. In addition, we provide several complementary techniques that enhance the quality of the extracted surface:

- Adaptive isosurface extraction optimized for the GPU programmable geometry pipeline
- Improved resulting surface quality through quadratic root-finding while maintaining lower extraction grids for memory saving

- Strict on-demand extraction of isosurface

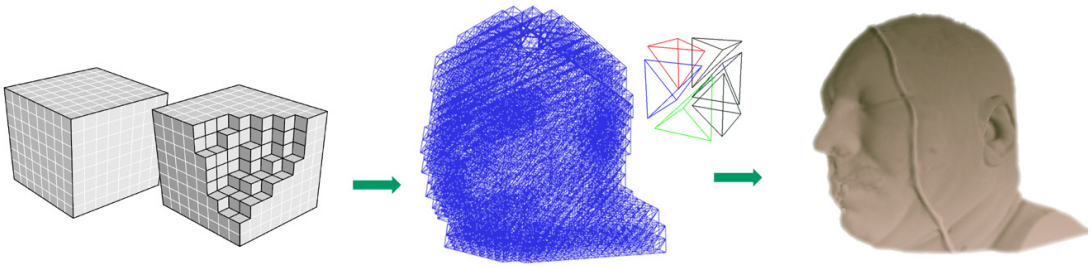


Figure 2. ***Isosurface extraction pipeline.*** *We start by voxelizing an implicit grid (left) into discrete cube cells. We then convert that to tetrahedral representation (center) and perform marching tetrahedra to extract polygonal surface for a given isovalue (right).*

Our isosurface extraction pipeline (Figure 2) proceeds as follows: we start by dynamically generating the voxel grid to cover our entire volume or a section of it. Using geometry shader and stream out feature, we tessellate the volume into tetrahedra on-the-fly. This allows us to adaptively generate and sample the grid, based on the demands of the application. Each input voxel position is dynamically computed in the vertex shader. Geometry shader then computes 6 tetrahedra spanning the voxel cube. As an optimization, we only generate tetrahedra for voxels containing the isosurface providing memory savings. Once we have the tetrahedra, we then use the marching tetrahedra algorithm to dynamically extract polygonal surface from our scalar volume consisting of material densities. In both passes for tetrahedral mesh generation and isosurface extraction we use geometry amplification feature of the geometry shader stage directly on GPU.

We utilize the efficiency of parallel processing units on the GPU more effectively by separating isosurface extraction into two passes. Given a set of input vertices, a geometry shader program will execute in parallel on each vertex. Given that each individual instance of this program is, in fact, serial on a given SIMD unit, we maximize each effective SIMD utilization by separating extraction into two phases - fast cube tetrahedralization and a marching tetrahedra pass, reducing serialization of each individual geometry shader instance. Thus, we first execute on all vertices in our grid in parallel, generating tetrahedra, and then execute on each tetrahedra in parallel, generating polygons. This also allows us the optimal balance between parallelization of polygonal surface extraction with efficient memory bandwidth utilization (the tetrahedra, exported by the first pass, consist of just 3 4-component floating point values).

9.3 Isosurface Extraction using Marching Methods

Our method is based on both the marching cubes and the marching tetrahedra algorithms for isosurface extraction. The domain in $\Re^3$ over which F is defined is tessellated into a grid at an arbitrary sampling density. In both methods, for each edge $e=(x_0,x_1)$ in the tessellation, we will evaluate $F(x_0)$ and $F(x_1)$; by the intermediate value theorem, if the signs of $F(x_0)$ and $F(x_1)$ differ, a isosurface vertex must intersect e. Considering all edges, we can connect vertices to form triangles.

Each possible assignment of signs to grid cells can be assigned a unique number. This is subsequently used to index into a lookup table, which will indicate the number and connectivity of isosurface triangles contained in the cell. In the marching cubes method, each cell has 8 vertices, and therefore there exist 28 possible assignments of the sign of F(x). Each cube typically produces up to 6 triangles (as described in [3]). Therefore a straightforward lookup table holds $28 \cdot 6 \cdot 3 = 4608$ entries. The size of this table presents an important consideration for parallel implementations on the GPU. The edge lookup tables are typically stored in SIMD-specific memory for efficient and coherent access by each shader program instance.

However, constructing large tables may result in additional register pressure. This would significantly lower the amount of parallel threads simultaneously running on the GPU. Smaller lookup tables ensure higher order of parallelization as the GPUs are able to schedule more threads due to a larger number of available registers. Furthermore, Ning and Bloomenthal [10] have demonstrated that MC can generate incorrect connectivity (even if the inherent ambiguities are avoided by careful table construction). Therefore, straight marching cubes polygonization would need to handle the undesirable topology in a special-case manner for each geometry shader invocation, increasing serialization for each instance of the program and thus significantly impairing performance of the resulting algorithm.

The marching tetrahedra method, by its use of a simpler cell primitive, avoids these problems. There exist only 16 possible combinations, emitting at most 2 triangles, and no ambiguous cases exist. This tiny lookup table allows effective use of the fast access SIMD registers and therefore results in much higher utilization of the parallel units on the GPU. One additional consideration for the related class of applications is that the cube is more intuitive for grid representation and adaptive sampling, with stronger correspondence to the original sampling. We note that the tetrahedron is an irregular shape and does not share these advantages. Straightforward tetrahedralization of a cube requires between 4 and 6 tetrahedra per cube, thereby requiring a corresponding factor of increase in the number of function evaluations required for the resulting mesh, if no sharing is done between primitives. In order to deal with this consideration, we introduce our hybrid method.

9.3.1 Hybrid Cubes-Tetrahedra Extraction

The GPU architecture excels at large-scale parallelism; however, we must remember that the programmable units perform in lock-step and therefore we must carefully parallelize our computations for maximum performance. The general strategy is to use marching cubes to exploit the additional information present in cubes as opposed to tetrahedra. As we mentioned earlier, straightforward tetrahedralization is a relatively complex program in GPU terms, thus would reduce thread parallelization. Rather than perform triangulation directly, it is preferable to adaptively tetrahedralize the input cubes. We perform final triangulation of the output surface using the simpler extraction operation on the tetrahedral grid.

Our method uses the following steps:

Pass 1: Domain voxelization

(1) Dynamically voxelize the domain
(2) Tessellate cubes into tetrahedra near the surface
(3) Output tetrahedra as points to stream out buffer

Pass 2: Marching tetrahedra

(1) Perform marching tetrahedra on generated tetrahedra
(2) Identify edges intersecting surface
(3) Fit a parabola on the edge and find root, by either:
    (a) Perform third function evaluation along edge
    (b) Use function gradients to estimate a parabola
(4) Output each isosurface triangle to a stream-out buffer for later re-use and rendering or straight to rasterization for immediate results

Voxelize Input Domain. In many cases (for example with volumetric data generated with medical imaging tools or physical simulations) the input data itself is specified on a regular (cubic) grid. Therefore, from the perspective of reducing function evaluations (corresponding to texture reads on the GPU) it is most practical to evaluate the function exactly at those points, and generate output triangles accordingly.

Although tetrahedral meshes provide a straightforward and efficient method for generating watertight isosurfaces, most preprocessing pipelines do not include support for generating tetrahedral meshes directly. Furthermore, we would like to support isosurface extraction on dynamic meshes and thus wish to generate tetrahedra directly on the GPU (for example, for particle or fluid simulations).

We start by rendering a vertex buffer containing n^3 vertices, where n is the grid size. Using automatically provided primitive ID, we generate voxel corner locations directly in the vertex shader. Subsequent geometry shader computes the locations of the voxel cube corners. We can then evaluate isosurface at each voxel corner.

Cube Tetrahedralization. In the first pass' geometry shader we tessellate each cube voxel containing the surface into at most 6 tetrahedra. We can either re-use already computed isosurface values, or, in order to reduce stream out memory footprint, simply repeat evaluation of tetrahedra corners in the next pass. Prior to tetrahedralization we compare the signs of the voxel corners to determine whether a given voxel contains the isosurface. Using the geometry shader's amplification feature, we dynamically generate tetrahedra for only those voxels which contain the isosurface. We output tetrahedra as point primitives into stream out buffers, typically storing only the (x, y, z) components of tetrahedra corner vertices for efficient use of stream-out functionality.

Adaptive Isosurface Refinement. We have a number of options for adaptive polygonization of the input domain. We can perform an adaptive subdivision of the input grid, during the first pass of domain voxelization. While sampling the isosurface, we can select to further subdivide the voxel refining the sampling grid in order to detect new isosurface parts that were missed by the original sampling grid. This uses straightforward octree subdivision of the input 3D grid and will generate smaller-scale

tetrahedra for the new grid cells on the finer scale, if the original voxel missed the isosurface.

We can additionally add adaptive tetrahedra refinement in the subsequent pass during cube tetrahedralization at very little cost. This allows us to generate new sampling points inside the existing grid cells where the isosurface has already been detected by the previous pass. In the 6-tetrahedra tessellation used, each tetrahedron shares a common longest edge along the diagonal of the cube. At the cost of one function evaluation at the edge midpoint, we can perform a longest-edge subdivision for each of the 6 tetrahedra. We emit only those of the resulting 12 tetrahedra that contain the surface, which are a subset of those previous discarded whose signs differ from the sign of the center point. By performing one additional evaluation and at most 6 comparisons, we can perform a two-level adaptive simplification. Note also that as the subdivision is always to the center shared edge, the tetrahedra of this cell will be consistent with those of neighboring cells and avoid cracks in the output surface.

Marching Tetrahedra. Using the *DrawAuto* functionality of DirectX10 we render the stream out buffer as point primitives in the next pass, performing the marching tetrahedra algorithm in the geometry shader stage. We identify the edges of the current tetrahedron which contain the output surface. As stated, MT is preferable for GPU implementation due to the significantly reduced lookup table sizes and output complexity. However, by the use of our hybrid method that reuses function evaluations from the initial cube grid, we can avoid redundancy, and by our adaptive subdivision step, we significantly reduce the number of primitives generated. Thus, our hybrid method is able to utilize the strengths of both methods as adapted to the GPU pipeline.

Fit a parabola along edges. While the Marching Tetrahedra rules identify which edges will result in an output vertex, they do not specify where along that edge the vertex will lie.

For an edge $e = (x_1, x_2)$, the vertex v_e will fall along the line segment $x_1 + (x_2 - x_1)t, t \in [0,1]$, and ideally, the choice of t is such that $F(v(t)) \approx 0$, so as to best approximate the isosurface. We can therefore consider finding the optimal vertex as a root-finding problem over t. The traditional approach uses the (linear) secant method:

$$t = \frac{-F(x_1)}{F(x_2) - F(x)} \qquad (1)$$

We have found, however, that significantly better visual quality results from using the inverse quadratic interpolation method (more familiar as a step of Brent's method [11]). This method fits a parabola to the function, and estimates the root by solving a quadratic equation. We illustrate the difference between the two methods in Figure 3. Our quadratic root-finding method performs faster than applying an additional level of subdivision, and in fact frequently produces better results than a corresponding increase in the sampling density. We provide two methods to fit such a parabola, according to the demands of the particular application.

Option 1: Perform a third evaluation per edge. The simplest way to fit a parabola along the edge is to perform a third function evaluation $F(x_3)$. The three points will then uniquely define the parabola. For a first approximation, we use the result of the secant method for x_3. Solving the quadratic defined by the points, we have:

$$v = \frac{F(x_2)F(x_3)}{(F(x_1)-F(x_2))(F(x_1)-F(x_3))}x_1 + \frac{F(x_1)F(x_3)}{(F(x_2)-F(x_1))(F(x_2)-F(x_3))}x_2 + \frac{F(x_1)F(x_2)}{(F(x_3)-F(x_1))(F(x_3)-F(x_2))}x_3$$

Because this additional evaluation is only performed along edges which are known to output a vertex, we are able to selectively restrict these additional evaluations to locations in which they are useful, as opposed to increasing the function grid size, which would lead to superfluous evaluations.

As we optimize the grid tessellation in previous phases, an additional evaluation is not a bottleneck in our implementation, and thus provides additional quality with no performance penalty. Should this be a limitation for certain applications, we propose an alternate method that avoids additional evaluations.

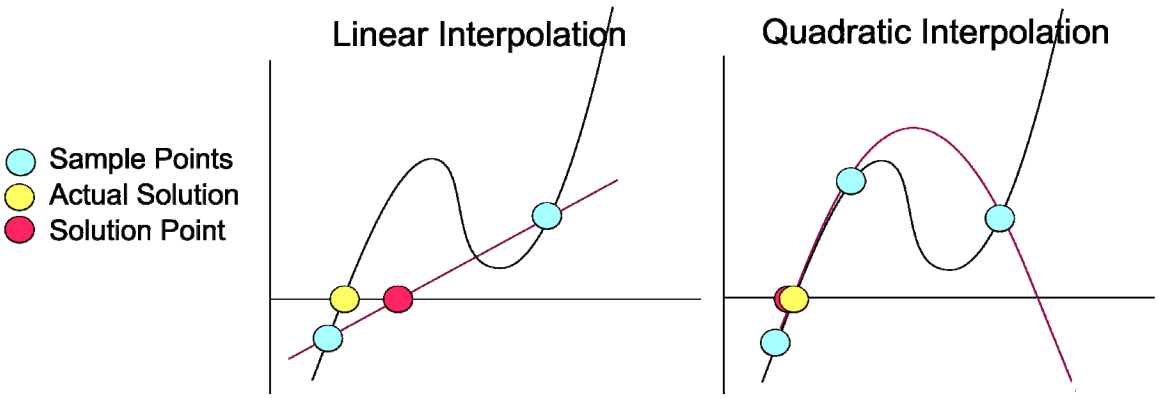


Figure 3. ***Linear vs. Quadratic Root-finding****. The traditional approach to finding the intersection point of the isosurface is to linearize the implicit function, and solve for the root (left). By taking an additional sample so as to fit a parabola, and finding a root of that parabola (right), we can make a better approximation of the actual zero-crossing.*

Option 2: Estimate parabola with gradients. Frequently, the function gradient $\nabla F(x)$ is either known, or can be computed easily along with $F(x)$. With static volume textures, ∇F is often computed beforehand, and stored with $F(x)$ in a single RGBA texture. Similarly, with common implicit functions, such as metaballs, the additional work required to compute the gradient is minimal. Finally, many applications will already be required to evaluate the gradient for use as a shading normal. In all such cases, we can use the gradients to estimate a parabola, obviating the need for the additional function evaluation.

We seek a parabola that interpolates both endpoints, and makes the best least-squares approximation to the gradients at each endpoint. We can restrict the problem the to line $v(t) = x_1 + (x_2 - x_1)t$, defining $F'(x)$ as the directional derivative:

$$D_{x_2 - x_1} F(x) = \nabla F(x \cdot (x_2 - x_1))$$

The class of parabola interpolating the endpoints, and its derivative, is:

$$F(t) = F(x_1) + (F(x_2) - F(x_1) - b)t + bt^2 \quad (2)$$

$$F'(t) = F(x_2) - F(x_1) - b + 2bt \quad (3)$$

We seek a choice of b that minimizes the least-squares error between the function derivatives relative to the actual derivatives, where the relationship is given by the following system of equations:

$$F'(x_1) \approx F(x_2) - F(x_1) - b \quad (4)$$

$$F'(x_2) \approx F(x_1) - F(x_{12}) + b \quad (5)$$

The least squares solution is the average of both solutions, or $b = \frac{1}{2}(F'(x_2) - F'(x_1))$.We can now solve Equation 2 directly using the quadratic equation, which is guaranteed to have exactly one root in the interval. Note that if $F'(x_2) = F'(x_1)$, this is equivalent to performing the linear secant method.

Storing Extraction Results for Subsequent Re-Use. We can intelligently generate isosurface on-demand (either as a function of the implicit domain changes or when the user modifies isovalue, using our material editor interface). After computing marching tetrahedra, we can output isosurface triangles into a GPU stream-out buffer for re-use in the later passes or even storage on-disk. This capability is a critical feature of our method that is allowed by the latest GPU pipeline. While the extraction already runs at real-time rates, the actual frame rate perceived by the user will be dramatically faster, as most frames are able to reuse the geometry from the stream out buffer. This frees up GPU resources for additional computation, such as, for example, combining high quality direct volume rendering with isosurface rendering for better context guidance during medical training simulations (as seen in Figure 1). Furthermore, we utilize the extracted polygonal surface to improve the rendering speed of our volumetric renderer. We render the isosurface faces as the starting positions for ray casting in the direct volume rendering algorithm (as an optimization for ray casting). The isosurface can also be rendered directly into a shadow buffer for shadow generation on the surrounding organs.

9.4 Direct Volume Rendering

To provide context to our isosurface visualization, we render the surrounding data directly. This is achieved by casting rays from the viewer, through each pixel of the screen, and sampling the volume data at a constant rate. We refer the reader to the seminal direct volume rendering papers by Drebin et al. [12] and Levoy [13] for more in-depth discussion.

Our implementation of direct volume rendering is a GPU ray casting method based on the work of Krueger and Westermann [14]. The algorithm begins by calculating ray directions in normalized texture coordinate space. This is performed by rendering the bounding box with a per-vertex color equal to the texture coordinate at that corner. First, the back faces of the bounding box are rendered to a screen sized texture to determine the exit point $Coord_{back}$ of the ray for each pixel. Subsequently, the front faces of the bounding box are rendered to texture to determine the entry point $Coord_{front}$. The ray direction in texture coordinate space for each pixel can then be calculated as $Coord_{back} - Coord_{front}$. Ray marching is then performed in a pixel shader by marching a ray from $Coord_{front}$, advancing by a fixed step size, for n steps. n is the number of steps required to advance to the exit point ($length(Coord_{back} - Coord_{front}) / step\_size$).

At each ray marching step, the absorption of color by the material at the current volume location of the ray must be accounted for. The color of the current location in the volume is modulated by the accumulated opacity of the ray and the opacity of the location. The result is added to the accumulated color. The following equations model describe this operation:

$$C_a = C_a + (1.0 - \alpha_a)\alpha(x)C(x) \quad (6)$$

$$\alpha_a = \alpha_1 + (1.0 - a_a)\alpha(x) \quad (7)$$

where $C(x)$ and $\alpha(x)$ are the color and opacity at volume location $\vec{x}$, and C_a and α_a are the accumulated color and opacity for the ray.

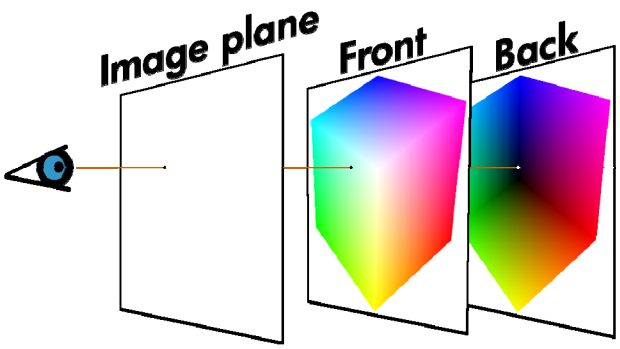


Figure 4. ***Ray directions*** *are calculated in texture coordinate space from the texture coordinates of front and back faces of bounding geometry.*

9.4.1 Incorporating Isosurface for Ray-casting Optimization

The purpose of volume rendering the data surrounding the extracted isosurface is to provide context. In our application, we treat the extracted surfaces as opaque. Thus, we can safely terminate our rays at the isosurface. This is achieved by rendering the

extracted isosurface's 3D texture coordinates into the exit point texture $Coord_{back}$. An example exit point texture is provided in Figure 5.

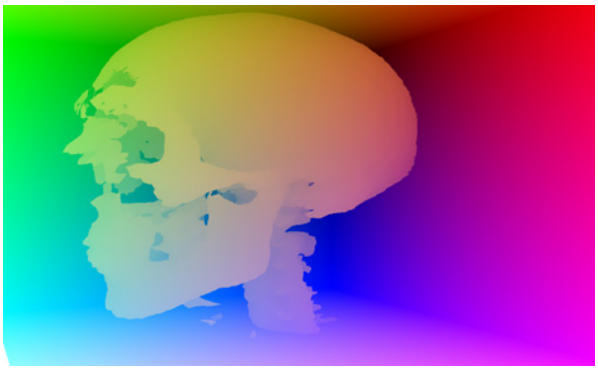


Figure 5. ***Incorporating the isosurface into the volume rendering*** *is achieved by rendering the isosurface into the exit point texture..*

This straight-forward integration of the extracted polygonal surface as the ray exit points results in unpleasant aliasing artifacts for the volume rendering. This is caused by the fact that during the ray casting computation, while computing each ray intersection for a given marching step, the new exit point may be simply missed. The isosurface coordinates may not lie exactly on the sample point of a marching ray. This will cause banding artifacts along the interface between volume rendering and isosurface rendering. To counteract the last incorrect sample which will be inside the surface, we weight it by the fraction of the step that is outside of the surface (Figure 6).

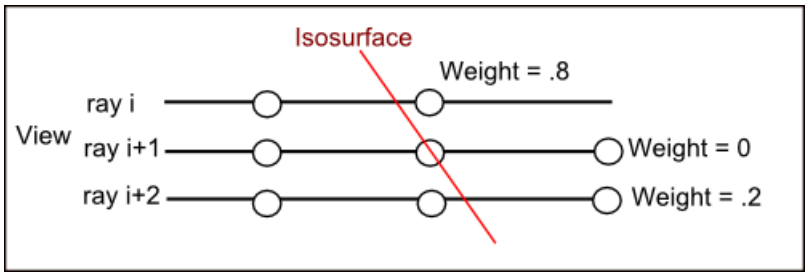


Figure 6. Using a fixed sampling rate requires the last sample to be weighted by the fraction of the last step that is outside of the isosurface.

9.5 Results and Conclusions

We collected results for continuous isosurface extraction and volume rendering for all seven sections of the visible human dataset. Each section contains 2563 samples on a regular grid. This results in 576 MB 3D textures memory footprint, storing the density and gradient information for each dataset portion. This dataset can be rendered in its entirety on the GPU. However, due to input data resolution discrepancy we chose to render the subsets as separate datasets. We use several high resolution off-screen buffers for rendering ray marching front and exit points as well as some intermediate information, taking up 120 MB of V-RAM. The stream-out buffer used for storing the polygonal surface for the extracted isosurface is 85 MB.

Isosurface Extraction Results and Analysis. Previous methods for GPU-based isosurface extraction have been forced to use contrived implementations to escape the limitations of the earlier programmable graphics pipelines. Such methods, while often performant, are complex to re-implement and modify, and typically do not support re-use without significant effort. We have shown that with the availability of latest generation GPU architectures, flexible and reasonable implementations of marching methods are possible to implement taking advantage of the massive parallelism available on the GPU, while maintaining optimal performance characteristics. We found that our hybrid method of dynamic domain voxelization following by a tetrahedralization pass results in high-performance and high-quality results. In our development of this system, we have explored various types of methods, including using standard marching cubes or marching tetrahedra directly. We settled on the final hybrid algorithm presented here after extensive testing and analysis of performance bottlenecks. The direct implementation of marching tetrahedra requires large amounts of redundant isofunction computations, reducing parallelization. The marching cubes performance was strongly reduced by the large look-up table sizes, as well as extraneous computations for incorrect topology fix-up. Therefore, this hybrid approach proved to be an excellent tradeoff between the two.

An earlier implementation performed the tetrahedral tessellation and surface extraction in a single pass. However, this severely limited parallelism. The difference in running time between a cube that is culled, and one that is both tetrahedralized and used for isosurface extraction in a single pass, creates a significant bottleneck for GPU resources. Instead, by moving the marching tetrahedra computation into another pass, we are able to fully utilize GPU SIMD programmable units parallelization for voxelizing the domain and generating tetrahedra near the surface. Similarly, the marching tetrahedra pass exhibits the same advantages.

All timing results include rendering cost and were collected on a Windows Vista PC with AMD Athlon 64 X2 Dual Core 2.4GHz, 2GB RAM and ATI Radeon HD 2900 XT graphics card.

Dataset	Grid	Time	Extracted Faces	Continuous Extraction FPS	Face/sec
Head	64^3	112 ms	114 K	8.93 fps	1 M
	32^3	21 ms	24 K	46 fps	1.2 M
	16^3	4.9 ms	5.5 K	203 fps	1.1 M
Thorax	64^3	129 ms	177 K	7.71 fps	1.3 M
	32^3	23 ms	38 K	43 fps	1.6 M
	16^3	5 ms	7.8 K	200 fps	1.6 M
Abdomen	64^3	143 ms	186 K	9.98 fps	1.3 M
	32^3	25 ms	40 K	39 fps	1.6 M
	16^3	5.5 ms	8.2 K	180 fps	1.5 M

Table 1. ***Timing results*** *for continuous isosurface extraction. See also the supplementary video for more results.*

While a common approach to smoothing extracted isosurfaces is to use linear root-finding; our quadratic root-finding approach produces significantly higher visual quality results (Figure 7). In our example surface detail quality is measured by surface smoothness. However, for different datasets, using this approach results in higher amount of surface details recovered (thus not necessarily a smoother surface). This additional computation has very minimal overhead by requiring less than 30 additional scalar ALU operations in the marching tetrahedra extraction shader, and an additional texture fetch, and as such having very slight impact on performance. Note that the quality of quadratic root-finding for recursion depth $d = 8$ exceeds that for linear root-finding for several additional levels of subdivision $(d = 10)$.

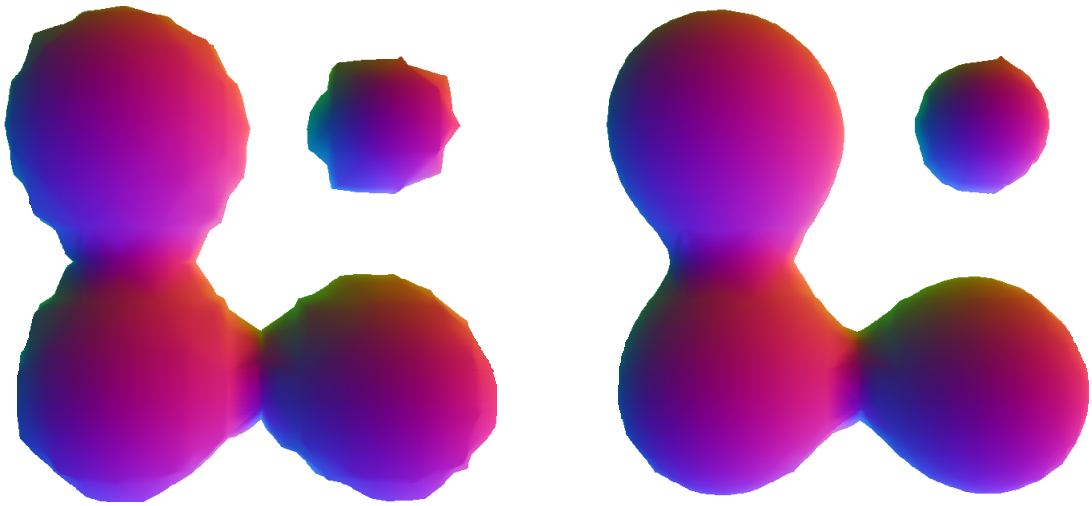


Figure 7. ***Results comparison of Root-finding Methods****. Linear secant method (left) results in surface missing important fine-grain detail (shown by the rough silhouettes of the smooth input metaballs). Quadratic root finding (right) results in significantly higher detail surface (and thus, smoother in this case) as compared to the linear secant method.*

Volumetric Rendering via Ray-Casting Results. Our ray caster is implemented utilizing Shader Model 3.0 functionality. We dynamically compute per-step lighting results integrating the resulting illumination using on the order of 600 ray marching steps for each ray. We found that the limiting factor for the ray caster performance is both the application resolution (resulting in higher fill rates for higher resolution) and the amount of steps taken for each ray computation. Integrating isosurface mesh as our ray exit points allows us significantly speed up the resulting rendering, often on the order of magnitude in speed for given viewpoints.

Conclusions and Future Work. We have presented a set of approaches that allow interactive exploration of medical datasets in real-time. Our technique is based on combining direct volume rendering via ray-casting with isosurface extraction on GPU. The latter takes advantage of the programmable DirectX 10 pipeline for dynamic surface extraction in real-time using geometry shaders. We have optimized our algorithm to take advantage of the massively parallel GPU architecture, while attuning it to the strengths and the constraints of this model. Combining isosurface with direct volume rendering allow for surface properties as well as for the context of tissues surrounding the region and gives better context for navigation. Our resulting application is both easy to use and results in high frame rates.

9.6 Acknowledgements

We would like to thank Daniel Szecket for his work on the user interface used for our system. We also extend special thanks to Dan Abrahams-Gessel and the Game Computing Applications group, as well as the anonymous reviewers for their help, suggestions and review of this work.

9.7 References

[1] WESTERMANN, R. AND ERTL, T. 1998. Efficiently using graphics hardware in volume rendering applications, in: SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques, ACM Press, New York, NY, USA, 1998, pp. 169–177.

[2] ROTTGER, S., KRAUS, M. AND ERTL, T. 2000. Hardware-accelerated volume and isosurface rendering based on cell-projection, in: VIS '00: Proceedings of the conference on Visualization '00, IEEE Computer Society Press, Los Alamitos, CA, USA, 2000, pp. 109–116.

[3] LORENSEN, W. E., AND CLINE, H. E. 1987. Marching cubes: A high resolution 3D surface construction algorithm, in: Computer Graphics (Proceedings of SIGGRAPH 87), Vol. 21, Anaheim, California, 1987, pp. 163–169.

[4] SHIRLEY, P. AND TUCHMAN, A. 1990. A polygonal approximation to direct scalar volume rendering, SIGGRAPH Comput. Graph. 24 (5) (1990) 63–70.

[5] PASCUCCI, V. 2004. Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping, in: Proceedings of VisSym 2004.

[6] KLEIN, T., STEGMAIER, S. AND ERTL, T. 2004. Hardware-accelerated reconstruction of polygonal isosurface representations on unstructured grids, pg 00 (2004) 186–195.

[7] GOETZ, F., JUNKLEWITZ, T. AND DOMIK, G. 2005. Real-time marching cubes on the vertex shader, in: Proceedings of Eurographics 2005.

[8] JOHANSSON, G. AND CARR, H. 2006. Accelerating marching cubes with graphics hardware, in: CASCON '06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research, ACM Press, New York, NY, USA, 2006, p. 39.

[9] SILVA, C., COMBA, J., CALLAHAN, S., AND BERNARDON, F. 2005. A survey of GPU-based volume rendering of unstructured grids, 17th Brazilian Symposium on Computer Graphics and Image Processing.

[10] NING, P. AND BLOOMENTHAL, J. 1993. An evaluation of implicit surface tilers, IEEE Comput. Graph. Appl. 13 (6) (1993) 33–41.

[11] PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. A. AND FLANNERY, B. P. 2002. Numerical Recipies in C++, Cambridge University Press.

[12] DREBIN, R. A., CARPENTER, L. AND HANRAHAN, P. 1988. Volume rendering, Computer Graphics 22 (4) (1988) 65–74.

[13] LEVOY, M. 1988. Display of surfaces from volume data, IEEE Computer Graphics and Applications 8 (3) (1988) 29–37.

[14] KRUEGER, J. AND WESTERNMANN, R. 2003. Acceleration techniques for GPU-based volume rendering, IEEE Visualization (2003) 287–292.

[15] KNISS, J., PREMOZE, S., HANSEN, C., SHIRLEY, P. AND MCPHERSON, A. 2003. A model for volume lighting and modeling, IEEE Transactions on Visualization and Computer Graphics 9 (2) (2003) 150–162.

[16] ENGEL, K., KRAUS, M. AND ERTL, T. 2001. High-quality pre-integrated volume rendering using hardware accelerated pixel shading, Workshop on Graphics Hardware.